CHALMERS | UNIVERSITY OF GOTHENBURG

A Decision Framework on Refactoring Architectural Technical Debt: Paying Back in Modularity
- An Industrial Case Study

Master of Science Thesis in the Programme Software Engineering

ERIK SIKANDER
NIEL MADLANI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, November 2015

A Decision Framework on Refactoring Architectural Technical Debt: Paying Back in Modularity
- An Industrial Case Study

ERIK SIKANDER
NIEL MADLANI

Supervisor: ANTONIO MARTINI
Examiner: ERIC KNAUSS

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

Department of Computer Science and Engineering
Göteborg, Sweden November 2015

Acknowledgements

First of all we would like to thank Antonio Martini for being our supervisor at the university, helping us with every step of our research by keeping us on track and helping us when we struggled. Furthermore we would like to thank Peter, Jesper, and Mattias who were our supervisors at Ericsson. They helped us get the information we needed regarding the company and its products as well as guided us to the right people if the information was not known to them. Lastly, we would like to thank the architects of the components analysed, together with the developers that we interviewed for helping us to get the information needed allowing us to conduct this research.

Erik Sikander & Niel Madlani

Vocabulary

ATD: Architectural Technical Debt (see Section 2.2.2)

DWM: Developer Work Month (see Section 4.2.1)

FOI(s): File(s) of Interest. The FOIs are the files that method of the study analyses to draw conclusion regarding if they should be modularized (see Section 2.1.3)

LOC: Line(s) of Code

MCC: McCabe Cyclomatic Complexity (see Section 2.2.1.1)

NRC: Non-Refactored Component (see Section 2.1.1)

PG: Product Guardian (see Section 2.1.2.1)

RC: Refactored Component (see Section 2.1.1)

Refactoring: rewriting code to make it better.

TD: Technical Debt (see Section 2.2.1)

Abstract

Technical debt refers to sub-optimal solutions during software development where there is a trade-off between short-term and long-term goals. Lately there has been a few studies which identifies technical debt, however most of the do not estimate the interest which is associated with the identified debt. Knowing how much interest is being paid allows the developers to make informed decisions of what will benefit the development. One example is knowing if a onetime cost of a repaying the debt outweighs the cost of paying the interest of that debt. This would mean that the repayment can be seen as an investment for future development. This thesis aims to develop a decision framework that can be used when deciding if part of a component would benefit from being modularized into a new component or framework to repay a debt. To accomplish this, the study developed two methods that are used by the decision framework. The first method is to find out if the analysed part of the components would benefit from such a modularization. The second method estimates how much effort can be saved by doing a modularization. It was found that for the first method, a measurement system which analysed the component's source code was a good approach in deciding if a modularization would be beneficial. For the second method an approach which used data regarding the current effort distribution to estimate the effort saved by modularizing was chosen. The result of combining the two methods was found to be an adequate decision framework which provides useful information in the decision if to modularize part of a component or not.

Contents

Chapter 1

Introduction

As software evolves, the quality often decay if the developer do not actively work against this decay (Hassaine, 2012). The decay originates from implementations that are not optimal for what they are set out to solve. In keeping the quality high, software companies has a lot to gain in the form of less expensive development and lower maintenance costs. The concept of source code having a so called Technical Debt (TD) has in recent years revived attention from both the industry as well as from the research community. With this attention and awareness there has risen a need to be able to measure and control this debt.

The term Technical Debt originates from an experience report written by Ward Cunningham back in 1992. In it he describes a debt within code as not-quite-right code:

> *"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. [...] The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt."* (Cunningham, 1992)

Nowadays TD has become more well-know and is used as a metaphor for the financial consequences that are associated with a quick-and-dirty solution which is often described as a trade-off between short-term goals and long-term goals. TD is similar to the financial debt, where one has to pay interest. The interest which is generated by the TD is in the form of extra effort, resulting in an increased maintenance and evolution cost (Kruchten et al., 2013; Nord et al., 2012). The choice can be made to repay the debt by refactor the affected parts or to continue pay the interest on the debt. However, the debt will increase with further development on the already existing debt and thereby the interest will increase as well if left unmanaged (Nord et al., 2012). However, TD does not only refer to source code implementations but also refers to immature design, incomplete documentation, and unfinished testing during the software life cycle as well (Guo and Seaman, 2011). Immature design, or Architectural Technical Debt (ATD), is regarded as sub-optimal solutions within architecture in order to support the business goals of the software (Martini et al., 2015). Examples of ATD are: dependency violation; non-uniformity of patterns and policies; code duplication (non-reuse); and temporal properties of inter-dependent resource. Another concept which is part of the ATD is modularity. Modularity means that changing one component will have minimal effect on other components which are connected to the ATD via reuse and dependency violation.

Technical Debt can have a large impact on the effort spent on both the maintenance and the continued development of a component due to the interest it generates. In order to control the accumulation of TD there is a need to identify its location, its size, its interest, and how it can be repaid. During the last couple of years there has been an increase in the number of tools that can be used to detect TD. Most of these tools analyses source code either statically or dynamically (Gat and Heintz, 2011; Schumacher et al., 2010; Nord et al., 2012; Antinyan et al., 2014; Heitlager et al., 2007). However, there has only been a few papers which have tried to identify and estimate the interest that the TD hasF. One such paper is that of Nugroho et al. (2011) which, based on statically analysing the source code, gives a component a rating. This rating is then used to predict the interest on the whole component and the cost of repaying the TD.

Even though there are some researches that are trying to estimate the interest, there are still some uncertainties in how they should be used as well as where and when they are applicable. A recent study by Martini et al. (2014) emphasized the need of identifying an optimal time to repay the interest on ATD before it gets too high. This is closely related to what Zazworka et al. (2014) emphasize, that TD should not be repaid if the cost of the repayment outweigh what will be saved by doing the repayment itself. Finding this balance between keeping the interest as low as possible and not wasting resource on unnecessary repayments has not been researched as much as other aspects of TD. This study aims to fill part of this gap by developing a decision framework that can be used to help decide if a component would benefit, in the form of effort, from being refactored. More specific, if a part of component would benefit from being modularized into its own component or framework. With modularization this study refers to extracting the complexity away from the original component, thereby hiding it and minimizing the impact it has on the rest of the source code. This allows it to be reused across multiple components as well as having fewer dependencies to other components.

To do this, this thesis conducted a case study to answer the following question:

> **RQ:** What are suitable methods to use when deciding on if to modularize a component?

Although, in order to answer the research question RQ in a more structured way, it was divided into two sub-research questions:

> **RQa:** How to decide if a component needs to be modularized?
> **RQb:** How to estimate the effort saved by repaying TD?

These questions led the study to design two methods, one for identifying if there is enough complexity to justify a modularization, and one for estimating if the effort saved will outweigh the effort of doing the refactoring.

The study's results indicated that the decision framework and its two methods provide enough information for deciding if a part of a component would benefit from being modularized or not. Furthermore, it was found that using a measurement system, following the ISO/IEC 15939:2007 standard, which measured the complexity of the source code statically, was a good method to find out if part of a component has TD that can be repaid by a modularization. Lastly, it was found that using data regarding the current effort distribution of the analysed component, it is possible to estimate how the effort will change after a refactoring.

The main contribution that this study has is the decision framework. The information given by the two methods provided enough information for the decision framework to be able set up guidelines whether or not to modularize a component, depending its TD and interest. Thereby, the decision model is able to help developers in their decision if they should modularize part of a component or not. This helps answer the main research question RQ.

Furthermore, the study contributes with a measurement system which analyses the source code statically to find out if a specified part of a component would benefit from being modularized based on its TD. This was done by using measures regarding complexity and effort. It was found that the measurement system were able to identify the part of a component that would benefit by being modularized, as well as the parts that did not within the research setting. This helps answer the sub-research question RQa.

The last contribution by this study is a method for estimating the effort that can be saved by a refactoring, or with other words, how much less interest is paid after the refactoring. This method also takes into account the double maintenance which occurs after the refactoring is completed where older releases are still being maintained. The study also explores a method to more accurately estimate the effort on those cases where a similar modularization had been done previously. This helps answer the sub-research question RQb.

To summarize the contributions:

- A decision framework helping developers decide if a component needs to be modularized (RQ)
  - A measurement system which can help in the decision if a component will benefit by being modularized (RQa)
  - Calculation model for estimating the amount of effort that can be saved by refactoring (RQb)

The remainder of this thesis is divided into six chapters. Chapter 2 outlines the background information needed to understand the study and consists of a domain background and a theoretical background. Chapter 3 describes the methods used by the study to answer the research questions. Chapter 4 describes the result of the study. Chapter 5 will describe the evaluation used to evaluate the decision framework presented in Chapter 4. Chapter 6 consists of discussions concerning the study. Chapter 7 concludes the important findings of the study.

Chapter 2

Background

This chapter describes the background which this study is based upon. Firstly, it describes the setting which the study was conducted in. This includes the domain which was used in order to draw the study's conclusions as well as the different stakeholders that have an association to the study. Secondly, it describes the theoretical background that the reader will need to be able to fully understand the study and the reasoning, decisions, and conclusions that it makes.

2.1 Domain Background

The study was conducted at Ericsson which is a large world leading telecom company with more than 110,000 employees. The company is providing equipment, software and services to enable transformation through mobility. The study was conducted at Ericsson located at Lindholmen, Gothenburg. The site has around 2500 employees, whereas around 120 developers work in the department where the study was conducted. The department is further divided into 18 cross-functional teams where each team consist of around 6-8 members. These teams are responsible for the maintaining, analysing, testing, and designing of new features for their products. The department provided the study with access to the projects source code as well as documentation and knowledgeable personal. This allowed the study to analyse the source code regarding the ATD connected to modularization as well as analysing the LOC change history.

2.1.1 The Domain

The domain that this study was conducted in consisted of three entities, two components and one framework. The two components shared the same functionality, implemented separately in each of the components. At first the functionality was implemented in the same manner. However, one of the components refactored their implementation by modularize it into a framework. The *refactored component* (RC) uses the framework, while the *non-refactored component* (NRC) still uses its old implementation. The two components and their relation to the framework are illustrated in Figure 2.1. Note that the NRC does not use the framework. The study used this context to develop a decision framework to answer the main research question, RQ. This was done by applying the decision framework on the NRC component to predict the impact it would have if it was modularized to use the same framework as RC.

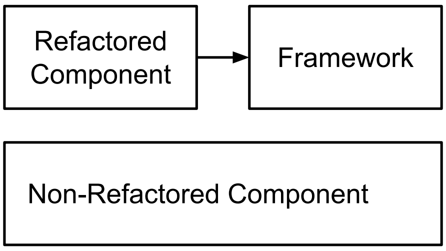

Figure 2.1: Illustration of the two components relation with the framework

The goals of the modularization and the creation of the components and framework were gathered through documentation provided by the company as well as interviews with key persons involved in its creation. The goals listed below is what the study used as a base for developing the different methods for answering the sub-research questions RQa and RQb and applying them on the NRC.

1. Reuse across multiple application and components
2. Reduce code complexity of code associated with the functionality
3. Reduce amount of application code in component related to the functionality
4. Shorter time to market
5. Minimize development and maintenance cost

2.1.2 Stakeholders

This section presents the stakeholders that both are affected and affects the results of the study.

2.1.2.1 Product Guardian

Product guardians (PG) are the ones who are responsible for the different frameworks and components. The product guardians provided technical knowledge in their respective domain of the components. They provided information of which parts of the components should be focused on, and which development teams are suitable to be participating in the interviews etc.

2.1.2.2 Developers

The developers of the teams working with the framework and/or components are another example of stakeholders. Not only are they affected by the result but they also provided the thesis with necessary data.

2.1.2.3 Managers

The managers are affected by the result of the study. They will be able to use the data in order to make well-grounded decision if to invest in a refactoring.

2.1.3 Files of Interest

The decision framework that this study has developed analyses part of a component. This part consist of what the study will refer to as a files of interest (FOIs), which are the source code files that are to be analysed regarding if they would benefit from being modularized. The FOIs may be identified differently for each component.

In this study the FOIs are the source code files associated with the functionality of the two components, RC and NRC. These FOIs are referred to throughout the study and are identified differently in the two components. The reason for this is that the components have different implementations as a result of the modularization RC had which was mentioned in Section 2.1.1.

In RC a source code file is identified as FOI if the source code file is associated to the framework. More specific, the C++ and H files which have a reference in the form of an '#include' to a file within the framework itself. This also implies that C++ files which implement a corresponding H file are identified as FOI as well. This identification process is illustrated in Figure 2.2. Example, if *A.h* has a reference to *FW.h* it is identified as an FOI. Furthermore, as the file *A.cpp* implements *A.h*, *A.cpp* will also be counted as FOI due to this implementation of a FOI. However, looking at the file *C.cpp* which only uses *A.h*, it will not be identified as a FOI since it is not implementing *A.h* nor is it using a framework file.

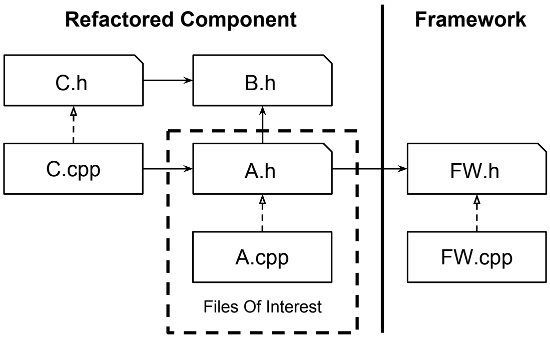

Figure 2.2: Illustration of how FOIs are identified in RC

In NRC a source code file is identified as a FOI manually. The reason for this is that the whole implementation of the functionality is within the component, it has no isolated reference point to do the automatic finding as it is done for the RC.

2.2 Theoretical Background

This section will describe the theoretical background of the study by giving the reader a description on TD, ATD and related concepts.

2.2.1 Technical Debt

As mentioned in the introduction, the term technical debt (TD) originates from an experience report written by Ward Cunningham back in 1992 (Cunningham, 1992). Nowadays TD refers to a sub-optimal solution in software development. This often happens when developers do quick and dirty implementation to be able to deliver more quickly (Fowler, 2003), which is like taking a loan for a short term goal with long term consequences if not paid back. In this way TD is similar to the financial debt, where one has to pay interest. The interest that is generated by the TD is in the form of extra effort, resulting in an increased maintenance and evolution cost (Kruchten et al., 2013; Nord et al., 2012). The choice can be made to repay the debt by refactor the affected parts or to continue to pay the interest on the debt. However, the debt will increase with further development on the already existing debt and thereby the interest will increase as well, if left unmanaged (Nord et al., 2012). The key difference between the financial debt and TD is that TD may in some cases never be repaid (Guo et al., 2014). In fact, when a software retires the TD connected to that software will retire as well (McConnell, 2011; Guo et al., 2014). Technical debt does not always have to be about poorly written code. TD can, for example, emerge from lack of documentation, lack of testing, bad architecture and so on.

2.2.1.1 Types of TD

In this study mainly two types of TD metrics will be used. These are McCabe Cyclomatic Complexity and Halstead Complexity.

McCabe Cyclomatic Complexity Back in 1976 McCabe (1976) wrote an article where he presented a complexity measure, known as McCabe Cyclomatic Complexity (MCC). This measure is based upon the number of linearly independent paths that a program can take, where each path adds to the complexity. Other papers such as Antinyan et al. (2014) have put it in a more concrete context, where they define it as usage of certain programming statements. However, there is a difference between papers as well as tools on what statements should be count towards the cyclomatic complexity. Antinyan et al. (2014) for example does not count *case*-statements towards their complexity while the tool Lizard (Yin, 2015) does. The statements that this study will count towards the MCC are: *if*, *else*, *while*, *for*, ||, &&, *case*, *default*, *return*, *goto*, *continue*, and *break*.

Halstead Complexity Halstead complexity was introduced by Maurice Halstead in 1977 (KRIS, 2001). Halstead's metrics are complexity metrics which are calculated from the operators and operands (Verifysoft, 2010). There are five metrics in total, these are *Halstead volume*, *Halstead difficulty*, *Halstead effort*, *Halstead time to implement*, and *Halstead delivered bugs*. All these metrics uses the operators and operands in the implementation analysed. More specific, they are using the total number of operators (N_1), number of unique operators (n_1), total number of operands (N_2), and number of unique operands (n_2). This study will use the same definition of operators as Verifysoft (2010). Examples of an operator are "=", "+", and "==". As for operands, this study counts anything that is not an operator as an operand. That is for example variable names, numbers, strings, and data types.

Halstead volume (V) is calculated by the program length times the 2-base logarithm of the vocabulary size: $V = (N_1 + N_2) * \log_2(n_1 + n_2)$ (Verifysoft, 2010). As the name suggest it describes the volume of an implementation, or how much information needs to be processed by the reader to understand the implementation. It also implies that using the same operand multiple times confuses the reader more than using more unique operands less frequently.

Halstead difficulty (D) is calculated as the proposition between total operators as well as the ratio between total operands and unique operands: $D = \frac{n_1}{2} * \frac{N2}{n2}$ (Verifysoft, 2010). This indicates that the Halstead difficulty increase more each time an operand is reused compared to having unique operands used seldom.

Halstead effort (E) is calculated by multiplying the Halstead volume with the Halstead difficulty: $E = V * D$ (Verifysoft, 2010). This defined as the effort required if the implementation code would be rewritten.

Halstead time to implement (T) is calculated by dividing the Halstead effort with 18: $T = \frac{E}{18}$ (Verifysoft, 2010). Halstead found that dividing with 18 was a good estimate of the amount of time in seconds it would take to re-implement the implementation.

Halstead delivered bugs (B) was originally calculated using $B = \frac{E^{2/3}}{3000}$ (Verifysoft, 2010). The metric itself is a prediction of number of bugs within the implementation. However, later paper such as Coleman et al. (1994) found that using Halstead volume gave a more accurate prediction of the number of bugs in an implementation. The formula was therefore revised to: $B = \frac{V}{3000}$.

Example A small example where the following code has its Halstead difficulty calculated:

int foo = x + x + 1;

This small piece of code would have 4 unique operands (int, foo, x, and 1) and 5 total operands (int, foo, x, x, 1). The number of unique operators would be 3 (=, +, and ;) and 4 total operators (=, +, +, and ;). Halstead difficulty calculation with these values would be $\frac{3}{2} * \frac{5}{4} = 1.875$.

2.2.2 Architectural Technical Debt

Architectural Technical Debt (ATD) is one of the many different TD types. ATD, or immature software design, is regarded as sub-optimal solutions within architecture in order to support the business goals of the software (Guo and Seaman, 2011). A sub-optimal solution is when a shortcut is taken in the architecture. ATD has also been defined as a software design that no longer fits the intended purpose (Zazworka et al., 2011), and as an imperfection of the software design that has a negative impact to the maintenance (Zazworka et al., 2013). According to Li et al. (2015) ATD is caused by decision on architecture that compromises some internal quality aspects, for example maintainability. But, such study lacks; reliable industrial studies; architecture anti-patterns; and studies involving the TD management process (Li et al., 2015).

Identifying ATD

Martini et al. (2015) mentions different types of ATD such as *non-uniformity of patterns and policies* and *code duplication(non-reuse).* Code duplication is a common ATD where similar code, or identical, is located in different components of the system where the code was managed separately which led to double maintenance (Martini and Bosch, 2015). A non-uniformity of patterns and policy is when the architecture is not consistence throughout the system. For example, name convention is applied in one part of them system but is not followed in another part of the system (Martini et al., 2015).

ATD can also be identified by checking if modularity violation exists in the architecture. A modularity violation takes place when the change of one component relying on the change from another component even though they belong to different modules and are supposed to be evolving individually (Wong et al., 2011). The ISO/IEC/IEEE 24765:2010 standard (ISO, 2010) defines modularity as *"The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components."*

Prioritize of Technical Debt Seaman et al. (2012) mentions that in the state-of-practice in software development and maintenance difficult decision has to be taken when prioritizing tasks during tight financial constraints. There are a few approaches that can be used in order to prioritize TD such as *cost-benefit analysis* and *analytic hierarchy process* (Seaman et al., 2012). The cost-benefit analyses approach has a"technical debt list" where each item in the list is undone and could cause a problem in the future. Each item has a description of why it needs to be done, where the problem occurs, and the estimation of the principal and interest. The principal refers to the effort required to complete the task (Seaman et al., 2012).

Chapter 3

Method

This chapter describes the method used to answer the research questions. It first describes why a case study was conducted and which tecniques was used. Then a description of getting an understanding of how a modularization implies for the RC component which is done through a case study. The last part is the design of the decision framework for answering the main research question, RQ.

3.1 Case Study

Case study is a research methodology for software engineering since it studies contemporary phenomenon in its natural context in order to understand how it interacts with the context (Runeson and Höst, 2009). This study uses the case study approach to investigate if a modularization would be beneficial (RQa) and how much effort can be saved by repaying the TD (RQb).Runeson and Höst (2009) defines four different purposes for conducting a case studies which can be used for different research methodologies and these are: exploratory; descriptive; explanatory; and improving. This study used exploratory since it seeking new insight and ideas in order to provide solution for new research. To provide a solution and find information, the study used mixed method which consist of both qualitative and quantitative research. The qualitative research was used to get more insight from the developers and PGs during interviews and surveys to find important data/information to be examined of the components. The quantitative research was used to collect the data from the components by statically analysing the source code.

In order to answer the research questions of this study the data has to be collected first. To collected the data, the study used different techniques from the qualitative and quantitative research methods. The techniques for qualitative research were interviews and survey, while for the techniques quantitative research were by statistically analysing the components. During the course of the study the authors conducted interviews, a survey, documentation analysis, and a static analysing tool at the case company. Together, these four techniques provided with technical knowledge and insight about how to answer the research questions.

3.1.1 Interview

The purpose of using interview is to collect data about phenomena which cannot be obtained through quantitative measure (Hove and Anda, 2005). Conducting interviews is an effective technique to get the participants' opinion and thoughts about something specific. Interview was a key part of this study since it provided with important data such as: information about the domain; the current problem; and what drove the developers to make the decision of modularizing the RC well as what part of the architecture they felt made the interest of the component too high.

Two types of interview methods were conducted in this study, semi-structured interview and structured interview. Conducting a semi-structured interview combines specific questions with open-ended questions where the latter is allows the interviewer to elicit unexpected information (Hove and Anda, 2005). The goal of structured interview was to find specific information and thereby specific questions are asked. For both interview types, the questions were constructed to have better possibilities to find interesting information which could help answering the research questions. The following list described the steps from finding participant too how the information was analysed:

1. Write relevant questions for the interview to be conducted.

2. Find relevant participants for the interview.

3. Interview the subject, where one asks the questions and the other notes everything down. Ask if the interview could be recorded.

4. Transcribe the information from the interview and analyse the information for interesting data.

5. Validate the data with the interviewees to see if they agree with the result.

3.1.2 Static analysis tool

To collect the data needed for the measures of the measurement system the study developed a tool which automatically calculates the measures for the measurement system. The tool was written in Python which allow it to be executed in multiple development environments. The reason for using a self-developed tool in contrary to existing tool is that it did not exist a single tool which gave all the values the measurement system needed.

The tool analyses the source code statically in order to calculate the MCC and Halstead bugs on each of the function in C++ files of the component. This is done by following the definition defined in Section 2.2.1.1 for MCC and Halstead. Furthermore, it uses the source code history in order to estimate the effort spread across the source code files by looking at the number of LOC changes.

3.1.3 Documentation analysing and Survey

The company provided documents which were used to gain knowledge about the domain the study will be investigating. Analysing these documents was an initial step since the authors had no previous knowledge about the domain. The information found was discussed between the two authors to see if further investigation was needed, such as an interview. Furthermore, several academic papers was analysed to understand the terminology and concept.

The basic idea of survey is to collect information from a group of people by sampling individuals from a large population (Linaker et al., 2015). In this study the survey was based on the descriptive method where the goal was to get the participants opinion on the RC and NRC. By doing a descriptive method it allows the authors of the study to strengthen their claim if a modularization is needed and if it would be beneficial.

3.2 Designing the Decision Framework

This section describes the method used for designing the decision framework in order to answer the main research question, RQ. The first step is to gather the relevant information, which this study did by interviewing experts in the subject as well as looking through documents. The second step is to use the information gathered and design a decision framework with the goal of it helping with the decision if a part of a component needs to be modularized. The created decision framework is then evaluated in Chapter 5.1.

3.2.1 Information Gathering

As mentioned in Section 3.2 , the first step was to gather the necessary information to find out what needs to be measured in terms of modularization. To do this, the study collected information of the RC where existing documents of the component was studied and by interviewing people involved in the modularization decision as well as the modularization itself.

In order to answer RQa documents were analysed to find information behind the reasoning for modularizing RC and what the expected outcome would be. Although, even if this would give reasoning on why RC was modularized, an interview was held with the people involved in modularization. The reason why an interview was conducted was so that the authors of this study would get a better understanding of the problems that led to the modularization and the reasoning behind them. The interview was done in a semi-structured where an experience developer and a PG of RC were interviewed.

Information regarding the sub-research question RQb was collected through interviews with software experts with experience in refactoring. This was to find out all the effort cost associated with a refactoring, or in the case of this study, a modularization.

3.2.2 Decision Framework Design

The second step as mentioned before is the designing of the measurement system. This is done by designing methods which answers the two sub-research questions, RQa and RQb. In order to answer RQa, where the goal is to investigate if part of a component needs to be modularized, the study designed a measurement system where the measures elicit the information collected as described in Section 3.2.1. The RQb is answered by creating an effort estimation equation which can be used to estimate the effort saved by refactoring.

3.2.2.1 Measurement System Design

The creation of the measurement system follows the ISO/IEC 15939:2007 standard (IEEE, 2009). The standard gives a good foundation of how a measurement system should be created by identifying both the activities that are involved as well as the tasks themselves. Together with guidelines from Staron et al. (2009) the study built its measurement system based on this standard. Therefore, this study will use the same definition of terms as it uses (IEEE, 2009). These definitions are:

Attribute: Property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means.

Analysis Model: Algorithm or calculation combining one or more base and/or derived measures with associated decision criteria.

Base Measure: Measure defined in terms of an attribute and the method for quantifying it.

Derived Measure: Measure that is defined as a function of two or more values of base measures.

Indicator: Measure that provides an estimate or evaluation of specified attributes derived from the model with respect to defined information needs.

Interpretation: Explanation of how to interpret the data of an indicator with a language that the intended user will understand.

Measurement Function: Algorithm or calculation performed to combine two or more vase measures.

Measurement Method: Logical sequence of operations, described generally, used in quantifying an attribute with respect to a specified scale.

3.2.2.2 Effort Estimation Equation

The creation of the effort estimation equation is done by analysing the information gathered from software experts as mentioned in Section 3.2.1. It does not analyse what factors affects the effort when it comes to refactoring, but instead how the factors are related to each other. This allows the study to combine these factors into an equation which is able to estimate the effort that can be saved by doing a refactoring, or in the case of this study, a modularization.

3.2.2.3 Decision Framework

By combining the two methods into a decision framework there will be enough information to make an informed decision if to modularize the analysed part of the component or not.

3.2.3 Evaluation

An evaluation of the designed decision framework is presented in Chapter 5. The reason for this is that the evaluation needs to be adapted to the decision framework which is presented in the Chapter 4.

Chapter 4

Result

This chapter presents the result of the study. It starts off by defining the two methods used to answer each of the two sub-research questions, RQa and RQb. The first method is a measurement system which helps the study to answer RQa. This is done by defining useful measures when deciding *if* a specific part of a component would benefit from being modularized. The second method helps the study to answer RQb by defining an equation which can be used to estimate the potential effort to be saved by doing a refactoring. This is followed by how to combine the two methods into a decision framework, to answer the main research question RQ.

4.1 Measurement System

This section describes the different measurements that the study will be using together with an explanation on why they were chosen. Then a coverage of the measurement system and the measures are presented followed by examples of the measures.

In order to answer RQa, the study developed a measurement system using the standard described in Section 3.2.2.1. The measurement system allows the study to analyse a combination of measurements which fits the purpose of finding *if* a part of a component would benefit from being modularized.

4.1.1 Measurement Decisions

From the initial interview (see Section 3.1.1) and analysing the documentation provided by the company, the study provided a list of the goals regarding the modularization described in Section 2.1.1. These goals were divided into two main categories, complexity and effort, and their contribution for answering RQa.

4.1.1.1 Complexity

Complexity measures was found to be a good choice when measuring modularity as modularizing a component is essentially hiding the complexity. There are a couple of metrics, such as McCabe and Halstead, which can be used to measure the complexity of a file or a function. The study aims to measure complexity across multiple files to analyse the FOIs, which the measures needs to use to be adapted. The complexity measures was chosen by interviewing people involved in the refactoring of the RC (see Section 3.1.1). In the interview they mentioned that they wanted to reduce the number of if- and for-statements in the code that was to be modularized. A measure that calculates the complexity is McCabe Cyclomatic Complexity (MCC), which is described in more detail in Section 2.2.1.1. Measuring the MCC provides information if this type of complexity can be hidden when it comes to, for example, if- and for-statements. However, McCabe is quite a controversial measure where it in some cases does not give an accurate value of the complexity of a function (Hummel, 2014). Therefore MCC was not the only complexity measurement which the study chooses to use. Halstead's metrics, more specifically Halstead delivered bugs metric, which is described in Section 2.2.1.1 was chosen to complement MCC. The metric estimates how many bugs exists in a function or file. Halstead was chosen because it calculates the operands and operators of an implementation in order to improve the readability. Therefore, by using this metric it will measure how well a modularization can hide the readability complexity.

4.1.1.2 Effort

The effort is an important aspect of refactoring where the main reason for refactoring is to save effort in the future. Effort has a direct correlation to the complexity as less effort is spent on the development of less complex code. This becomes a useful measure for answering RQa because if a component has TD and no effort is put into the component, then there is no interest being paid on the debt. The same goes for a part of a component, such as FOIs. It is important to take this into consideration; even if a component has a lot of TD it may not be worth repaying it since the interest could be less than the actual cost of doing the repayment. The measure for the effort in the developed measurement system is the LOC committed to the source control system. The reason why the measurement system uses LOC is to collect the data in a quantitative way in order to compare the FOIs with the rest of the code without any manually input data.

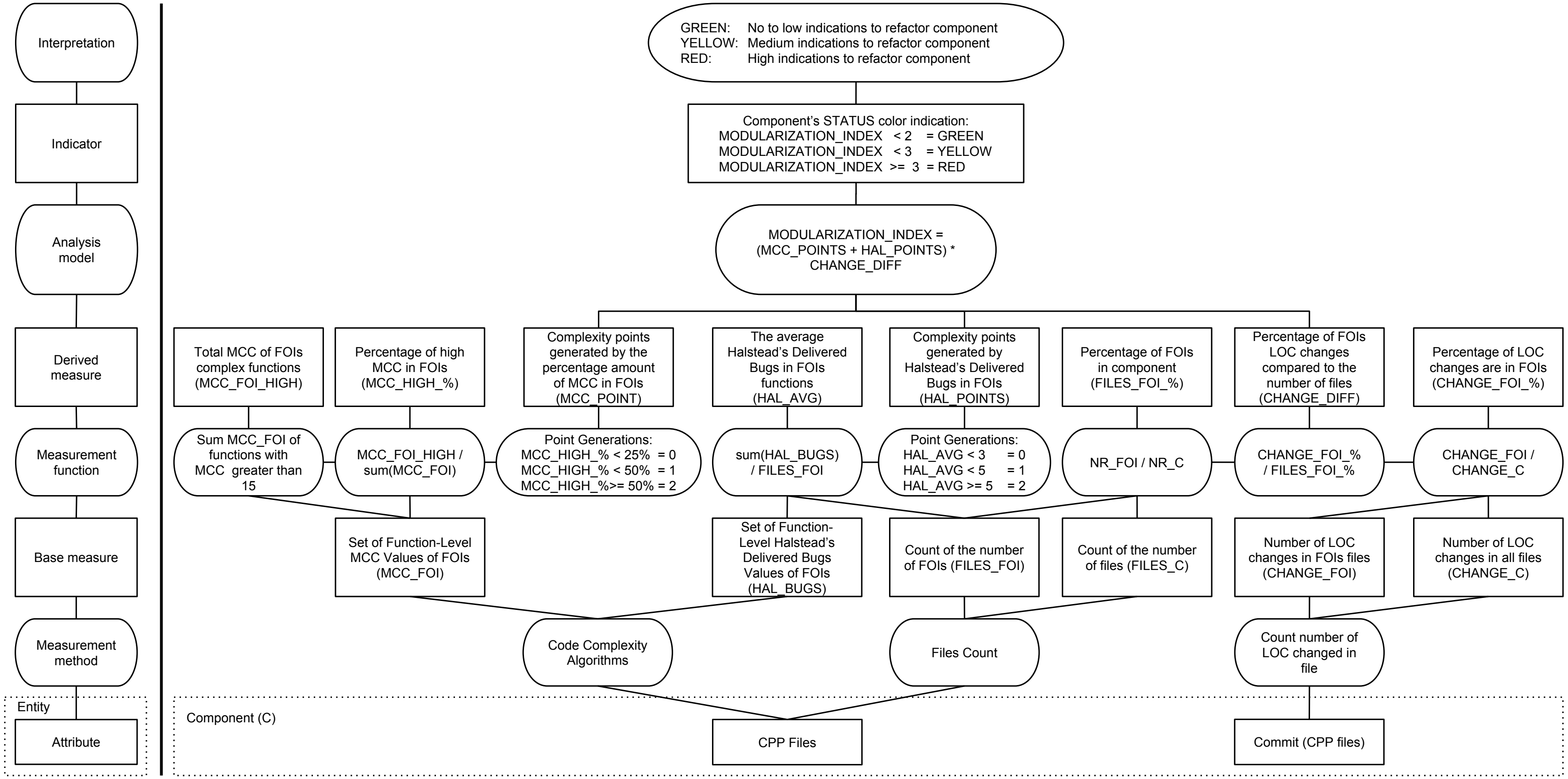
Interpretation
Indicator
Analysis model
Derived measure
Measurement function
Base measure
Measurement method
Entity
Attribute
GREEN: No to low indications to refactor component
YELLOW: Medium indications to refactor component
RED: High indications to refactor component
Component's STATUS color indication:
MODULARIZATION_INDEX < 2 = GREEN
MODULARIZATION_INDEX < 3 = YELLOW
MODULARIZATION_INDEX >= 3 = RED
MODULARIZATION_INDEX = (MCC_POINTS + HAL_POINTS) * CHANGE_DIFF
Total MCC of FOIs complex functions (MCC_FOI_HIGH)
Percentage of high MCC in FOIs (MCC_HIGH_%)
Complexity points generated by the percentage amount of MCC in FOIs (MCC_POINT)
The average Halstead's Delivered Bugs in FOIs functions (HAL_AVG)
Complexity points generated by Halstead's Delivered Bugs in FOIs (HAL_POINTS)
Percentage of FOIs in component (FILES_FOI_%)
Percentage of FOIs LOC changes compared to the number of files (CHANGE_DIFF)
Percentage of LOC changes are in FOIs (CHANGE_FOI_%)
Sum MCC_FOI of functions with MCC greater than 15
MCC_FOI_HIGH / sum(MCC_FOI)
Point Generations:
MCC_HIGH_% < 25% = 0
MCC_HIGH_% < 50% = 1
MCC_HIGH_%>= 50% = 2
sum(HAL_BUGS) / FILES_FOI
Point Generations:
HAL_AVG < 3 = 0
HAL_AVG < 5 = 1
HAL_AVG >= 5 = 2
NR_FOI / NR_C
CHANGE_FOI_% / FILES_FOI_%
CHANGE_FOI / CHANGE_C
Set of Function-Level MCC Values of FOIs (MCC_FOI)
Set of Function-Level Halstead's Delivered Bugs Values of FOIs (HAL_BUGS)
Count of the number of FOIs (FILES_FOI)
Count of the number of files (FILES_C)
Number of LOC changes in FOIs files (CHANGE_FOI)
Number of LOC changes in all files (CHANGE_C)
Code Complexity Algorithms
Files Count
Count number of LOC changed in file
Component (C)
CPP Files
Commit (CPP files)

Figure 4.1: The measurement system developed in the study based on the ISO 15939:2007 standard (IEEE, 2009)

4.1.2 Measurement Definitions

In this section a detailed definition is given for the measurement methods, measurement function, and analysis model seen in Figure 4.1. The figure visually explain all the different measures and indicator as well as which of them are connected, from base measures to indicator starting from the bottom.

4.1.2.1 Attributes

The developed measurement system analyses the C++ source files, not including the H files. This is because the complexity metrics that is used by the measurement system's measures are on a function level and the H files would not always generate usable data. Including them would mean that the data of the complexity would be contaminated with files that do not add to the measures, but still lower the average. Also, other measures that would be able to use the H files data could not be justifiable be combine with those that cannot. The measurement system also looks at the source code history to determine how the effort is spread out throughout the files in the component. Furthermore, the measurement system does not analyse the test files. Although, the test files do have some complexity to them, which could be argued to be a part of the FOIs, the test files are not always part of the core functionality which is what this measurement system is designed to assess.

4.1.2.2 McCabe Cyclomatic Complexity

This section defines the base measure and the derived measures in the measurement system that are associated with the MCC. The MCC is one of two complexity measures that are used by the measurement system to determine if the FOIs would benefit from being modularized. To do this, it uses the following base- and derived measures:

Base Measure - MCC_FOI This measure is a set of function-level MCC values in FOIs. How the MCC is calculated is defined in Section 2.2.1.1.

Derived Measure - MCC_FOI_HIGH The MCC_FOI_HIGH sums the MCC of the complex functions within the sets MCC_FOI. The threshold that determine if the MCC of a function is considered complex or not is not universally defined. McCabe (1976) suggested the threshold to be set to 10, but stated that the number was not, to quote, a "magical" one. Others papers such as Nugroho et al. (2011) and Heitlager et al. (2007) puts it as high as 50. Heitlager et al. (2007) define a different threshold where 50 is the last step, but defines a MCC of more than 20 to be complex, which is the threshold mentioned in Jones (1994). Antinyan et al. (2014) on the other hand uses a threshold of 15.

In this study a threshold of 15 was chosen due to the fact that previous studies at the same company used 15 as their threshold (Antinyan et al., 2014) with a successful result, as well as being within the norm of what is normally used. For example, if a function has a MCC of 10, then those 10 MCC will not be counted as complex MCC. But if another function has a MCC of 30, then those 30 MCC will be counted as complexity MCC. This shows that in this example 30 MCC out of total 40 MCC are complex MCC.

Derived Measure - MCC_HIGH_% Antinyan et al. (2014) defines this measure as *effective cyclomatic complexity percentage* or percentage of complex cyclomatic complexity in a file. In their paper it was used to calculate the percentage of MCC in a file. However, the developed measurement system aims to analyse on a set of file rather than a single file, where MCC_HIGH_% is the percentage of complex MCC in all the FOIs. Thereby giving $\text{MCC\_HIGH\_\%} = \frac{\text{MCC\_FOI\_HIGH}}{\text{MCC\_FOI}}$. Example, if there are 50 MCC total in all FOIs and of those, 30 MCC are marked complex. That would make 60% ($\frac{30}{50}$) of the MCC in all FOIs to be complex.

Derived Measure - MCC_POINTS This measure is used to generate points based on the percentage of complex MCC in the FOIs (MCC_HIGH_%). The thresholds used are those defined by Heitlager et al. (2007) for moderating risks (an MCC over 11) where it defines the threshold with regards to a file. In this study it was applied on a set of files, more specifically the FOIs. First off, a good file is defined by Heitlager et al. (2007) to have a complex MCC percentage of less than 25%. Therefore, 0 point is generated if the percentage of complex MCC in the FOI is less than 25%. Secondly, a file is defined as bad by Heitlager et al. (2007) if it has a complex MCC percentage of more than 50%. This means that from 25% to 49% this study will deem to be an intermediate value and thereby generate 1 point. Above 50% means it is very complex, and will therefore generate 2 points. These values are summarized in Table 4.1.

Threshold	Point
$< 25\%$	0
$< 50\%$	1
$\geq 50\%$	2

Table 4.1: The thresholds of how many points will be generated with regards to MCC_HIGH_%

4.1.2.3 Files Count

This section defines the base measure and the derived measures in the measurement system that are connected to the counting of files. The purpose is to distinguish the FOI files from non-FOI files to get an overview of how many files of the component are FOI and non-FOI. The measure is important since it is used to normalize other measures such as Halstead delivered bugs (see Section 4.1.2.4) and LOC Changes (see Section 4.1.2.5). The measures are:

Base Measure - FILES_FOI Count of the number of FOIs.

Base Measure - FILES_C Count of the number of files in the component, including the FOIs.

Derived Measure - FILES_FOI_% This measure is, in percentage, how many files in the component are FOIs. Example, if there are 10 files in the component and 2 of them are FOIs, then FILES_FOI_% would be equals to 20%.

4.1.2.4 Halstead Delivered Bugs

This section defines the base measure and the derived measures in the measurement system that are connected to the Halstead delivered bugs measures. Halstead is the second of the two complexity measures used by the measurement system to determine if the FOIs would benefit from being modularized. Halstead uses operators and operands to do calculations, which makes it associated with readability. The measures help with determining the overall quality of the program and predict the rate of error.

Base Measure - HAL_BUGS This measure is a set of Halstead derived bugs calculations on a function-level. How Halstead is calculated is defined in Section 2.2.1.1.

Derived Measure - HAL_AVG This measures sums the values of the functions in HAL_BUGS and divide it with the number of FOIs. This gives the average number of bug that is expected to exist in a FOI. For example, if there are 3 functions in all FOIs where the first function with Halstead delivered bugs is calculated as 0.5, the second function is calculated as 2, and the third function is calculated as 3.5. That would show that the HAL_AVG would be the average of these three functions, which is $\frac{0.5+2+3.5}{3} = 2$.

Derived Measure - HAL_POINTS The study did not find any paper which used a threshold for Halstead delivered bugs. Therefore, the thresholds defining the points generated by this measure was discussed and agreed with the PGs with respect to HAL_AVG. The lower threshold for zero points was set to 3 while the threshold for two points was set to 5. Therefore, one point is generated if HAL_AVG is between 3 and 5. These are summarized in the Table 4.2.

Threshold	Point
< 3	0
< 5	1
≥ 5	2

Table 4.2: The threshold regarding how many points will be generated to HAL_POINTS with regards to HAL_AVG

4.1.2.5 Files LOC Change History

This section defines the base measures and the derived measures in the measurement system that are connected to the files change history.

Base Measure - CHANGE_FOI Counts the number of LOC changes in FOIs the last x months, where x is defined when the data is gathered. Depending on what the user of the measurement system is looking for they can use different values. For example, if the component underwent a refactoring 6 months earlier, then they may want to look 6 months back as the effort distribution may have changed due to the refactoring.

Base Measure - CHANGE_C Similar to CHANGE_FOI but it looks at the whole component rather than just the FOIs. Thereby, it counts the total number of LOC changes in all files of the component, including FOIs.

Derived Measure - CHANGE_FOI_% The percentage of file changes occurred in FOIs.

Derived Measure - CHANGE_DIFF The derived measure CHANGE_DIFF is the normalized version of the derived measure CHANGE_FOI_%. The result gives a percentage of how much more or less frequent a LOC change in a FOI compared to a non-FOI. This is calculated through the equation: $\text{CHANGE\_DIFF} = \frac{\text{CHANGE\_FOI\_\%}}{\text{FILES\_FOI\_\%}}$. A CHANGE_DIFF of 100% indicates that the FOIs do not receive more LOC changes on average than a non-FOI. A CHANGE_DIFF of less than 100% indicates that FOIs receive less LOC changes on average than a non-FOI. A CHANGE_DIFF of more than 100% indicates that FOIs receive more LOC changes on average than a non-FOI. For example, if 20% of the files are FOIs and 30% of the LOC changes are in FOIs, then CHANGE_DIFF will have a value of 150% ($\frac{30\%}{20\%}$). This means that a FOI is 50% more likely to get a LOC change compared to a non-FOI.

4.1.2.6 Analysis Model and Indicator

The Analysis model uses the points from the two complexity measures, MCC_POINTS and HAL_POINTS, as well as the effort in form of the likelihood of a LOC change in a FOI compared to a non-FOI (CHANGE_DIFF). By using the three measures the analysis model calculates a modularization index (MODULARIZATION_INDEX), which indicates if the FOIs would benefit from being modularized. The higher the index value, the bigger the need for a modularization. The index is calculated using the following equation:

$$\begin{aligned}\text{MODULARIZATION\_INDEX} = {} & (\text{MCC\_POINTS} + \text{HAL\_POINTS}) \\ & * \text{CHANGE\_DIFF}\end{aligned} \tag{4.1}$$

The two complexity measures, MCC_POINTS and HAL_POINTS, are added together as a general complexity point of the FOIs. The rationale behind that is that when analysing the source code the authors of this study found that a function with complex MCC did not imply a high value of Halstead delivered bug or vice versa. This observation was strengthened by the result of Zazworka et al. (2014) where different complexity method which they analysed did not overlap. The more general complexity points are then multiplied with the percentage of effort that is spent on FOIs in order to account for the effort. This shows that even if the FOIs are very complex and almost no effort is spent on them, the modularization index may not be that high. The arguments are based on the discussion in McConnell (2011) and Guo et al. (2014) regarding source code that do not change does not pay any interest, even if it has TD.

The indicator of the measurement system generates GREEN, YELLOW, or RED. The GREEN indicates that the measurement system did not find any indication that the FOIs are generating enough extra effort to the development for it to benefit from a modularization. YELLOW indicates that the measurement systems found indications that the FOIs generates interest in the form of effort and may benefit from being modularized, and developers should thereby be aware of it. RED indicates that the measurement system find strong indications that the FOIs are generating interest in the form of effort which would warrant a need for modularization, meaning that the developers should look into it. This indicator is directly linked to the RQa.

The colors that is generated by the indicator depend on the modularization index value, MODULARIZATION_INDEX, which has three thresholds (see Table 4.3). By doing this calculation (see Eq. 4.1) it will give the MODULARIZATION_INDEX value which will indicate if the FOIs would benefit from being modularized. If the MODULARIZATION_INDEX is less than 2, then the FOIs are marked as GREEN. If the MODULARIZATION_INDEX is equal or greater than 2 but less than 3, then the FOIs are marked as YELLOW. If the MODULARIZATION_INDEX is greater or equal to 3, then the FOIs are marked as RED.

For example, if the complexity measures generates 3 points, where one generates 2 points and the other generates 1 point, and the effort spent is 100% then the indication will be RED by following then Eq. 4.1. Another example would if the measurement generates 2.4 points, where one generates 1 points and the other generates 2 points, and the effort spent is 80% then the indication will be YELLOW by following Eq. 4.1.

Threshold	Point
< 2	GREEN
< 3	YELLOW
≥ 3	RED

Table 4.3: The thresholds of what color the measurement system will generate as an indicator based on the MODULARIZATION_INDEX

4.1.3 Measurement System Examples

In order to fully understand the measurement system this section will provide a couple of examples. These examples will be minimal, thereby skipping the measures on the lower level and focus more on the derived measure, which are connected to the analysis models and upwards (see Figure 4.1).

4.1.3.1 Example - Some Complexity Indicating YELLOW

In this example there are 20 C++ files where five files are marked as FOIs, which gives a FOI percentage of 20%. Out of the total LOC changes on the files, 40% are within the FOI, which gives CHANGE_DIFF a value of 200%. Furthermore, 25% of the MCC of the FOIs will be marked as complex generating a MCC_POINTS of 1. Lastly the average number of Halstead delivered bugs within the FOI files is 1 generating zero points for HAL_POINTS. These values are summarized in Table 4.4.

Measure	Values
CHANGE_DIFF	200%
MCC_HIGH_%	25%
MCC_POINTS	1
HAL_AVG	1
HAL_POINTS	0
MODULARIZATION_INDEX	2
STATUS	YELLOW

Table 4.4: Measurement system example data.

Given these data the measurement system would in this case indicate that the system does have some TD and that it is generating interest to the development. Although, it is generating interest it may not be enough to justify a modularization of the FOIs, and thereby the measurement system indicator is YELLOW. However, in order to make such decision further analysis will be required, for example the method defined in Section 4.2, in order to be sure that a modularization is needed.

4.1.3.2 Example - High Complexity Indicating GREEN

In this example there are 100 C++ files where five files are marked as FOIs, which gives a FOI percentage of 4%. Out of the total LOC changes on the files 1% are within the FOI, which gives CHANGE_DIFF a value of 25%. Furthermore, 50% of the MCC of the FOIs will be marked as complex generating a MCC_POINTS of 2. Lastly the average number of Halstead delivered bugs within the FOI files is 6 generating two points for HAL_POINTS. These values are summarized in Table 4.5. Given this data the measurement system in this case would indicate to the user that the system does have TD and that it is generating interest to the development. However, as the FOIs are not changed that often, the measurement system indicates that the FOIs may not need to be modularized at the moment. This may change

Measure	Values
CHANGE_DIFF	25%
MCC_HIGH_%	50%
MCC_POINTS	2
HAL_AVG	6
HAL_POINTS	2
MODULARIZATION_INDEX	1.2
STATUS	GREEN

Table 4.5: Measurement system example data.

if the FOIs start to be developed more frequently. Therefore, the measurement system's indicator is GREEN.

4.1.3.3 Example - Low Complexity Indicating RED

In this example there will be 100 C++ files where five files are marked as FOIs, giving a FOI percentage of 5%. Out of the total LOC changes on the files 20% are within the FOI, which gives CHANGE_DIFF a value of 400%. Furthermore, 10% of the MCC of the FOIs will be marked as complex generating a MCC_POINTS of 0. Lastly the average number of Halstead delivered bugs within the FOI files is 3 generating a HAL_POINTS of 1. These values are summarized in Table 4.6:

Measure	Values
CHANGE_DIFF	400%
MCC_HIGH_%	10%
MCC_POINTS	0
HAL_AVG	3
HAL_POINTS	1
MODULARIZATION_INDEX	4
STATUS	RED

Table 4.6: Measurement system example data.

In this example the measurement system indicator indicates RED. The reason for this is that the amount of effort that is spent on the FOIs will be increasing the interest that is generated by the TD on the FOIs.

4.1.4 Limitations of the Measurement System

The measurement system does not answer if a modularization of the FOIs will repay itself; it only indicates if a modularized implementation of the FOIs would be beneficial for further development. To be able to determine if a modularization does repay itself, the study created another method with this purpose which is presented in Section 4.2.

Users of the measurement system should be aware that the measurement system draws its conclusion from a group of files and determine their common characteristics where one bad file might hide among a couple of good file. Even though the measurement system indicates GREEN, it may in fact be worth looking into individual files, since there might be files that would benefit from being refactored on an individual level.

4.2 Effort Estimation Equation

One of the goals of the study is to answer RQb by estimating how much effort can be saved by refactoring and if a refactoring would be beneficial. To do this, the study needed to do a deeper investigation of what efforts are connected to a refactoring. The investigation was done through interviews with developers, PGs, and senior specialists. From the interviews the study obtained some key effort groups for the model calculation. The first group is the *old development effort* (E_1) which includes the effort spent on development before the refactoring. The second group is the *old maintenance effort* (E_2) which includes the effort spent on maintenance before the refactoring. The third group is the *new development effort* (E_4) which includes the effort spent on development after the refactoring. The fourth group is the *new maintenance effort* (E_5) which includes effort spent on maintenance after the refactoring. The last group is the *refactoring effort* (E_3) which includes the effort spent on refactoring.

To answer RQb these groups were combined into an equation (see Equation 3.2). The old development effort and new development effort defines the development effort gain from a refactoring. The old maintenance effort and new maintenance effort defines the maintenance effort gained from a refactoring.

$$E = (E_1 - E_4) * t + (E_2 - E_5) * t - E_3 \tag{4.2}$$

The equation 4.2 gives an overview of what needs to be in the equation, but the equation needs to be more specific. For example, it needs to be able to handle maintenance of the older version as well as the new version. Hence, the equation was expanded into the following equation:

$$\begin{gathered} M(x) = \begin{cases} \frac{r-1}{r}E_2 + \frac{1}{r}E_5 & \text{if } \frac{x}{m} < 1 \\ \frac{r-2}{r}E_2 + \frac{2}{r}E_5 & \text{if } \frac{x}{m} < 2 \\ \vdots & \\ \frac{r-(r-1)}{r}E_2 + \frac{r-1}{r}E_5 & \text{if } \frac{x}{m} < r-1 \\ E_5 & \text{otherwise} \end{cases} \\ E = (E_1 - E_4) * t + \Big(E_2 * t - \sum_{x=1}^{t} \big(M(x)\big)\Big) - E_3 \end{gathered} \tag{4.3}$$

The different variables mean the following:

- E: Effort Saved or Lost After t Months

- E_1: Effort of Development Before Refactoring in Months
- E_2: Effort of Maintenance Before Refactoring in Months
- E_3: Total Effort of Doing the Refactoring
- E_4: Effort of Development After Refactoring in Months
- E_5: Effort of Maintenance After Refactoring in Months
- t: Time After the Refactoring in Months
- m: Months Between Releases
- r: Number of Releases Being Maintained at a Single Point in Time

$M(x)$ is a function which calculates the maintenance associated with the component at month x after the release. The value returned is dependent on the number of releases being maintained and the time between releases. For example, if a release is maintained for 4 months and a new release comes every 2nd month ($m = 2$), then there would be 2 releases maintained at a single point in time ($r = 2$). The function uses this data and assumes that the maintenance effort is spent equally between the maintained releases. This imply that after the first release with the refactoring it is assumed that half of the maintenance effort is spent on the refactored release and the second haft is spent on the older non-refactored release (if the previous example data is used). Furthermore, when the non-refactored releases stops being maintained then the only cost will be the refactoring release. A more in-depth example can be found in Section 4.2.2.

The equation 4.3 itself is similar to that of Eq. 4.2. The difference being the variable used has been more defined and the way of calculating the new maintenance effort is expanded. The result, E, of the equation is effort difference between doing the refactoring or not at t months after the refactoring.

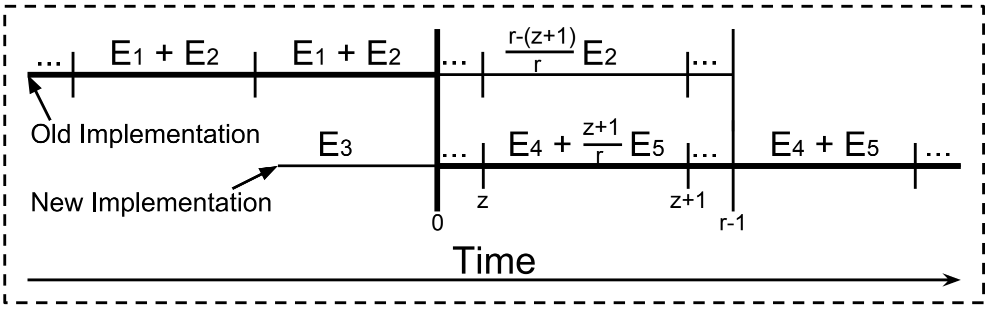

Figure 4.2: Illustration of how the variables are related in Eq. 4.3. Each vertical line represents a release while the horizontal lines represent time.

Figure 4.2 illustrate how the different variables relate to each other in a development time perspective. Each vertical line represent a time of a new release, where the thicker longer line represent the release from when the refactoring is introduced.

The thicker horizontal lines indicate main development as well as maintenance while the thinner lines represent only maintenance (E_2) or the refactoring (E_3). For in-depth description as well as how the variables are elicited see Section 5.1.2.3.

4.2.1 Developer Work Month

The study decided to present the calculation and result in the form of Developer Work Months (DWM). DWM is defined in this study as the amount of effort one developer generates each month. For example, if a developer works 22 days each month and get 6 hours of work done, then the effort in man-hours the developer generates in one DWM is $22 * 6$ which is equals to 132 man-hours each month. That illustrate that one DWM can be roughly translated to 132 man-hours.

4.2.2 Full Example

In order to fully understand the Equation 4.3 and its result, this section will present an example of how it is used and how the result can be interpreted, starting with the data gathering.

Let's say that there are 20 developers working full time on the development of the component being analysed. Furthermore, 20% of the LOC changes the last year have been on the files that is marked FOI. Therefore, $E_1 = 20 * 20\% = 4$ DWM/month. As E_2 is elicited in a very similar manner as E_1 with the difference of being the maintenance instead of development. Therefore, it will get the data from the number of developers working on maintenance and the LOC changes made in maintenance. In this example, this will be in a separate maintenance branch. For example, if 4 developers work full time on maintenance and around 50% of the LOC changes is in the maintenance branch of FOIs. This will give an E_2 of 1 DWM/month ($E_2 = 4 * 50\% = 2$ DWM/month).

E_4 and E_5 which are the development effort after a refactoring will most likely be estimated. This estimation can come from experience of people who have refactored a component before, or it can come from using data of a component that has already been refactored in a similar manner. In this example the data will be estimated by using data from another component. The data for E_4 and E_5 are therefore gathered in the same way as E_1 and E_2 were, but they are gathered from the component which has undergone a refactoring regarding the same functionality. Let's say that there are two developers working full time on the FOIs and one developer working full time on maintenance of the FOIs. By taking these numbers it would give: $E_4 = 2$ DWM/month and $E_5 = 1$ DWM/month.

The E_3 variable is different compared to the other E_x variables. It focuses more on estimating the development effort required for refactoring. For example, it is estimated that it will take 3 developers 5 months to refactor and this would give: $E_3 = 3 * 5 = 15$ DWM.

In this example, a release is maintained for 4 months after its release and a new release comes every second month. This means that 2 releases are being maintained at a single point in time. That gives $m = 2$ months and $r = 2$ releases.

Variable	Value
E_1	4 DWM/month
E_2	2 DWM/month
E_3	15 DWM
E_4	2 DWM/month
E_5	1 DWM/month
m	2 month
r	2 release

Table 4.7: Summary of the example data.

Table 4.7 summarize the data in this example so far. Now that all the data has been elicited, it is time to apply the equation 4.3.

$$
\begin{aligned}
M(x) &= \begin{cases} \frac{r-1}{r}E_2 + \frac{1}{r}E_5 & \text{if } \frac{x}{m} < 1 \\ \frac{r-2}{r}E_2 + \frac{2}{r}E_5 & \text{if } \frac{x}{m} < 2 \\ \vdots & \\ \frac{r-(r-1)}{r}E_2 + \frac{r-1}{r}E_5 & \text{if } \frac{x}{m} < r-1 \\ E_5 & \text{otherwise} \end{cases} \\
&= \begin{cases} \frac{2-1}{2} * 2 + \frac{1}{2} * 1 & \text{if } \frac{x}{2} < 1 \\ 1 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1.5 & \text{if } \frac{x}{2} < 1 \\ 1 & \text{otherwise} \end{cases} \qquad (4.4) \\
E &= (E_1 - E_4) * t + \Big(E_2 * t - \sum_{x=1}^{t} \big(M(x)\big)\Big) - E_3 \\
&= (4-2) * t + \Big(2 * t - \sum_{x=1}^{t} \big(M(x)\big)\Big) - 15 \\
&= 4 * t - \sum_{x=1}^{t} \big(M(x)\big) - 15
\end{aligned}
$$

Figure 4.3 and Figure 4.4 illustrates the variables relation to each other. The second of the two figure shows the values of the variables instead of the variable names. From these figures it is possible to see how the $M(x)$ function operates. During the first two months (depending on the m variable), half of the maintenance will be spent on the older release which uses the old maintenance effort (E_2) and the other half on the newly refactored version using the new maintenance effort (E_5). The reason for this is that there are two releases being maintained in this example. After the second release with the refactored implementation, this example spends all of the maintenance effort on the new maintenance effort (E_5).

Using the Eq. 4.4 it is possible to estimate the amount of effort saved after t number of months. Let's say that a manager want to know if restoring a component

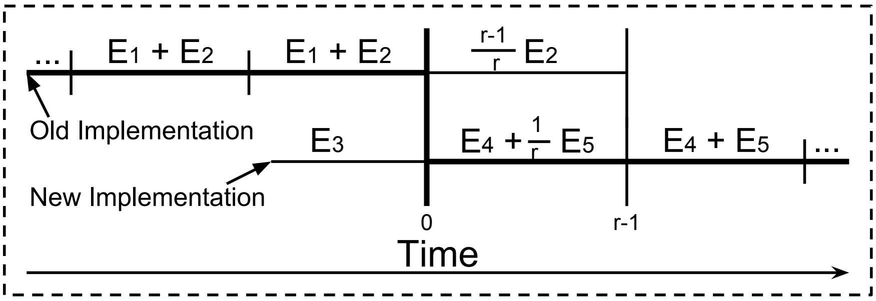

Figure 4.3: Illustration of how the variables are related in Eq. 4.4. Each vertical line represents a release while the horizontal lines represent time.

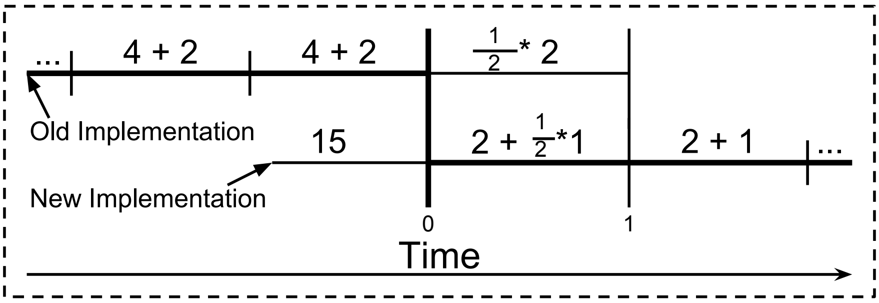

Figure 4.4: Illustration of how the variables are related in Eq. 4.4 with the variable values printed. Each vertical line represents a release while the horizontal lines represent time.

will help him reach his 12 month goal of increasing productivity. This can be calculated by setting $t = 12$. The calculation looks as the following:

$$\begin{aligned} M(x) &= \begin{cases} 1.5 & \text{if } \frac{x}{2} < 1 \\ 1 & \text{otherwise} \end{cases} \\ E &= 4 * t - \sum_{x=1}^{t} \big(M(x)\big) - 15 = \\ &= 4 * 12 - (1.5 * 2 + 1 * 10) - 15 \\ &= 48 - (3 + 10) - 15 = 20 \text{ DWM} \end{aligned} \tag{4.5}$$

From the calculation, the refactoring will save 20 DWM after 12 months. However, there are more information that can be extracted using the variables received. First off, by setting $E = 0$ the Eq. 4.5 can calculate the t time when the refactoring will repay itself after the refactoring is done. In this example it would be (assuming $t \geq 2$ as after 2 month the example draws full benefit for the new maintenance effort):

$$\begin{aligned} E &= 4 * t - \sum_{x=1}^{t} \big(M(x)\big) - 15 = \\ &= 4 * t - \big(1.5 * 2 + 1 * (t-2)\big) - 15 \\ &= 3 * t - 16 = 0 \\ &\implies t = \frac{16}{3} = 5.33 \text{ months} \end{aligned} \tag{4.6}$$

In this example, doing the refactoring, the developers would start saving effort after 5.33 months. This gives the developers and/or managers more information to use when deciding if a refactoring will be worth doing or not. For example, if the component will continue be developed the next couple of year, it can be seen as an investment to refactor the component since there will be a return in effort after 5.33 months. However, if the component is going to be replaced in the next 5 months, then according to the calculations from equation 4.5, the refactoring would not give a return in the lifespan of the component. Secondly, by doing the refactoring it is estimated that the developers will be spending 3 DWM/month ($E_1 + E_2 - E_4 - E_5$ or seen in Eq 4.5) less effort on the development and/or maintenance of the FOIs in the component. Thereby, it will allow the developers to work on other tasks or features. This is also the interest that is being paid by not refactoring the component. Lastly, the total effort spent on the refactoring is 16 DWM in this example. This includes the double maintenance effort of having to maintain two different versions of the component.

4.3 Decision Framework

By combining the two methods defined in Section 4.1 and Section 4.2 there is enough information to be able to make an informed decision if to refactor a component by modularizing it, and thereby answer the main research question, RQ. The way to combine the two models is illustrated in Figure 4.5. The figure illustrates a decision graph which can be used to make the decision if to refactor the analysed part of the component (the FOIs) based on the outcome of the two methods.

The first step of the decision model is apply the measurement system defined in Section 4.1 which will result in an indicator of GREEN, YELLOW, or RED based on the complexity of the FOIs and effort spent on the them. GREEN indicates that the measurement system did not find enough modularization violations that are draining effort from other development to warrant a refactoring by modularizing, and therefore no further analysis is needed. YELLOW indicates that some violations with interest were detected draining effort from the developers. However, the decision framework leaves it to the developer to choose if they want to go on with the effort estimation calculation or not, where the first is recommend by the researchers. If the latter is chosen, the developer should still pay some extra attention on the FOIs, as they were found to have some violations which had interest. In the Figure 4.5, this option is marked with a dashed line. RED indicates that the

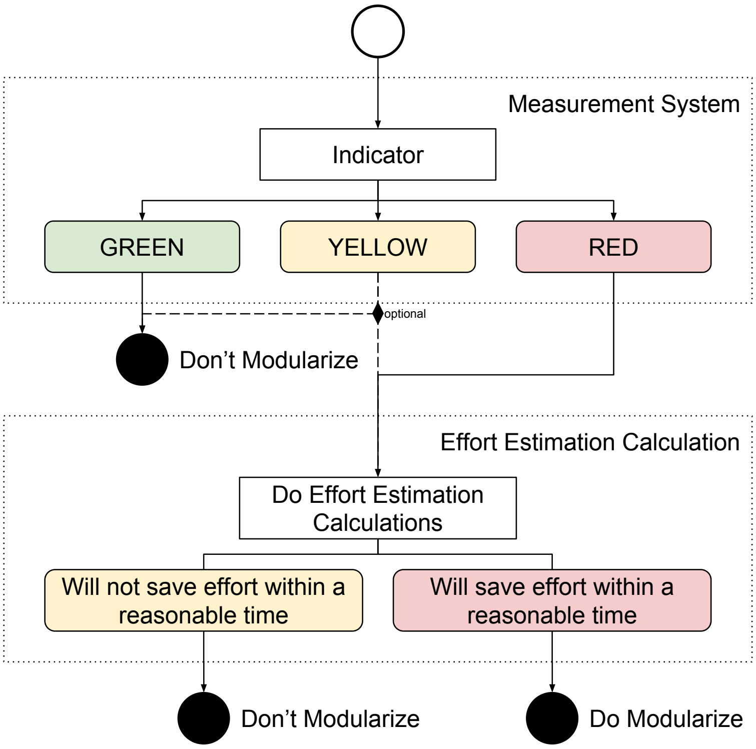
Measurement System
Indicator
GREEN
YELLOW
RED
optional
Don't Modularize
Effort Estimation Calculation
Do Effort Estimation Calculations
Will not save effort within a reasonable time
Will save effort within a reasonable time
Don't Modularize
Do Modularize

Figure 4.5: The decision graph of deciding if to refactor by modularizing

measurement system found violation that are draining effort from the developers and a closer analysis if a refactoring by modularization of the FOIs will repay itself within a reasonable future.

The second step is the effort estimation calculation which is defined in section 4.2. The outcome of this step is dependent on the analysed component's planned future. For example, if the component is to be replaced in 6 months, and the estimated time before the refactoring will start to repay itself in 7 months, it is not worth refactoring. However, if the component were to be developed past the 7 months, it is worth refactor the FOIs.

Example: Applying the measurement system on a component and its FOIs resulted in an indicator of YELLOW. This indicates that the measurement system leaves it to the developer to choose if to proceed with the effort estimation calculation or not. As recommended, this example will continue with the effort estimation calculation in order to see if there is effort to be saved by refactoring. The calculation indicated that by doing a refactoring, the developer on the component can expect effort being saved after 3 months. As the component is planned to be developed for at least 12 more month it is worth doing the refactoring by modularizing the FOIs.

Chapter 5

Evaluation

This Chapter present the evaluation of the decision framework presented in Chapter 4. This is done by first present the method used, followed by the result of the evaluation.

5.1 Evaluation Method

This section presents a method for evaluating of the decision framework defined in Chapter 4. The evaluation will be done by applying the decision framework on the two components, RC and NRC. The chapter will first present how the data was collected for the measurement system defined in Section 4.1. This is followed by how the data was collected for the effort estimation equation defined in Section 4.2. Finally, it presents the method used to validate the result of the decision framework.

5.1.1 Measurement System

To collect the data needed for the measurements of the measurement system the study developed a tool which automatically calculates the measures for the measurement system. This tool was written in Python which allow it to be executed in multiple development environments. The reason for using a self-developed tool in contrary to existing tool is that it did not exist a tool which gave all the values the measurement system needed.

The tool analyses the source code statically in order to calculate the MCC and Halstead delivered bugs on each of the function in the C++ files of the component. This is done following the definition for MCC and Halstead delivered bugs defined in Section 2.2.1.1. Furthermore, it uses the source code history in order to estimate the effort spread across the source code files by looking at the number of LOC changes.

5.1.2 Effort Estimation Equation

This section present the method used to gather the data required to be able use the equation presented in Section 4.2. This is done by first defining two of problems with the data collection and what the study did to solve these. Following that, there is a description of how the data was collected for the equation to do the calculation for the NRC.

5.1.2.1 Problem Size Scaling

One of the problems that the study faced was to be able to compare NRC with RC. The problem lies in the fact that the two components are not the same, they only share a functionality. This functionality is implemented to solve a problem, and there is a difference in form of the size of this problem between the two components, where the NRC's problem size is bigger than the RC's problem size. In order to make the study able to compare the two components, it needs a way of normalizing between them. More specific, the study needs a way to transform the data from RC to be able to fit the NRC by scaling up the data from RC. This is done by finding a way to get a ratio between NRC and RC using the following equation:

$$\text{PROB\_SCALE} = \frac{PROB\_SIZE\_NRC}{PROB\_SIZE\_RC} \tag{5.1}$$

The rationale behind this equation is that if the problem of NRC is twice as large, then the effort required when scaling from RC twice as much.

Example If the NRC using the shared functionality to handle 6 objects and RC use it to handle 3. The number of objects handled could be used to define the problem size difference. This would mean that the NRC has a problem of size 6 (PROB_SIZE_NRC = 6) and the RC has problem size to be 3 (PROB_SIZE_RC = 3). Using Eq. 5.1 gives a PROB_SCALE of 2. This means that the problem of NRC is twice as hard as that of RC.

5.1.2.2 Measuring the Effort of a Component

The company, where the study took place, did not have documentation specifying the exact amount of effort put into the development of a specific component and let alone the effort put on the FOIs, which are needed in order to do calculation regarding effort on FOIs. Therefore, other means of extracting that information had to be taken. The route the study decided on was to conduct interviews, which were held with the PGs and developers of both NRC and RC. More information regarding the interviews and their execution can be found in Section 5.1.2.3. During these interviews, questions were asked regarding the component as a whole instead of focusing on FOIs. The reason for this was because if the questions were asked directly regarding the effort spent on FOIs, then the interviewee would not be able to give a good answer. This is because they had trouble estimating the effort spent on FOIs as they did not exclusively work with those files. To get an estimation of the effort spent on FOIs, the study used the general effort spent on the component together with the percentage of LOC changes on FOI from the version control systems the last 12 months. For example, if the interviewee said that a total of 100 DWM was spent on the component the last month and the LOC changes in the source control system indicate that 10% of the changes were on FOIs, then the study estimates that $100 \text{ DWM} * 10\% = 10 \text{ DWM}$ was spent on FOI.

There are a couple of reasons why the study choose to look back in history of the files by a time period of 12 months. Firstly, looking back 12 months the effort will not be affected by any specific holiday. For example, if the study were to look back only six months then, depending on the time of the year, the developers may have been on vacation large percent of the time. This is avoided if looking back 12 months, because then the history data will account for most of the holidays. Secondly, it gives recent averages, meaning that if the way that the developers work with the component has changes recently the data is less affected then if the study were to use 24 months.

5.1.2.3 Data Collection

This section will describe how the data was collected by performing an interview and by analysing the source code. The data collected are the different variables that are used in Eq. 4.3.

Interview Design The interviews were conducted with the developers and PGs who have worked with RC and/or NRC. The purpose was to get an estimation on how many developers are working with the respective components. These interviews were conducted using a more structured approach. The steps mentioned in 3.1.1 were conducted for this interview.

Old Effort - E_1 and E_2 In this study the old effort data is gathered from NRC as that is the component that is being analysed regarding if it should be refactored (by modularization). This means that the information regarding the variables E_1 and E_2 are collected from developers and PGs with knowledge of NRC. The questions asked during the interviews regarding the data for E_1 variable (the development effort) were:

> How many developers are working on the development of NRC on a monthly basis?
>
> How many percent of their time is devoted to development of NRC?

The questions asked regarding the E_2 variable (the maintenance effort) were:

> How many developers are working on the maintenance of NRC on a monthly basis?
>
> How many percent of their time is devoted to maintenance of NRC?

The data gathered from the interviewees was derived using the method described in Section 5.1.2.2.

New Effort - E_4 and E_5 The data regarding the new effort was gathered from the RC as this component has already been modularized in the way the NRC is being analysed for. However, due to the fact that RC and NRC are different the data needs to be scaled according to Section 5.1.2.1. Although, before that, the data collected still needs to be derived according to Section 5.1.2.2.

The variables that are associated with the new effort are E_4 and E_5 which are the development effort and maintenance effort, respectively. The questions asked to get the data for E_4 variable were:

> How many developers are working on the development of RC on a monthly basis?
>
> How many percent of their time is devoted to development of RC?

And for the E_5 variable the following questions were asked:

> How many developers are working on the maintenance of RC on a monthly basis?
>
> How many percent of their time is devoted to maintenance of RC?

Refactoring Effort - E_3 The E_3 variable which is the estimated time it will take to refactor, which in this study is the time it would take to modularize NRC. This information was collated from the same people interviewed for the old effort variables in Section 5.1.2.3, the developers and PGs of NRC. The question that was asked to retrieve this data was:

> How much effort, in developers and months, would you estimate it would take to refactor the NRC with regards to the shared functionality?

Maintenance Variables - m and r Information regarding the maintenance time was collected from documentation. Although, the information may, for example, be in the form of "A release is maintained for 12 month and a there is a new release every 3 months". This means the variables may need to be calculated. From the example the m variable, which is the time between releases, is 3 and the r variable, which is the number of releases being maintained, would be $\frac{12}{3} = 4$.

5.1.3 Validation

Two types of validation were conducted for validating the evaluation. The first validation aims to check if the developers find that the component would benefit from being modularized, thereby help validate the result of the measurement system (RQa) and how much effort can be saved thereby help validating the result of both the measurement system and the effort estimation calculation (RQb). The second validation method aims to check if the decision framework would help to make a decision if a component needs to be modularized (RQ).

The first validation was conducted as a survey to get the developers opinion regarding the FOIs in the RC and the NRC. The process for choosing the developers was performed in two ways. The first one was by asking the PGs which teams were working with the RC respectively the NRC while the second one was by checking the source code history to find developers that had worked on the source code of the RC or the NRC. A pilot study was conducted where two developers were chosen as test subjects before the actual survey. The purpose was to make sure they understood the questions. Both of the developers understood the questions and were able to answer them. The survey contained questions regarding complexity, refactoring need and effort for the RC respective the NRC. The result from the developers would be calculated separately for each component and question in order to get an average value of their opinion. The result also separated in order to distinguish between those who had worked with the specific functionality and those who had not. The questions which were asked in the survey were:

1. Have you worked with the functionality in RC/NRC? (Yes/No)

2. On a scale of 1-10, where 1 is "Not complex" and 10 is "Very complex" , how complex is it to add/modify/remove source code of the functionality in RC/NRC?

   > - Example: "I find it really complex to add/modify/remove code in the functionality files or files dependent on them. I also find it hard to predict how the changes will affect the rest of the code. Therefore, I would say that it is of complexity 8" - Example answer: 8

3. On a scale of 1-10, where 1 is "Refactoring not needed" and 10 is "Needs refactoring" , how would you rate the need to refactor the functionality in RC/NRC?

4. On a scale of 1-10, where 1 is "Not a lot" and 10 is "A lot", how much effort do you think would be saved by refactoring, as mentioned in question 3, the functionality in the RC/NRC?

The second validation was an interview with the PGs of the two components to get there opinion regarding the result of the study. The purpose was to have a discussion on the result regarding the measurement system and effort estimation calculation as well as the data gathering. Therefore, the information shown to the PG not only consisted of the result, but also all the data gathered in order to make sure that the data itself do not contain any errors.

5.2 Evaluation Result

This section presents the results of the evaluation, using the methods from Section 5.1. First it will present the result of the measurement system applied on the two components, RC and NRC. Then the chapter will present the result of the effort estimation calculation for determining how much effort NRC can save by modularizing the FOIs. After the two methods, the result of the decision framework will be presented. Lastly, there will be a validation of the results.

5.2.1 Measurement System

This section presents the result from the measurement system, defined in Section 4.1, when applied on the two components, the RC and the NRC. This is done in order to evaluate and discuss the measurement system as part of RQa by presenting the result of the different measures as well as a short analysis of the results.

5.2.1.1 Measurement System Data

The data which the result uses is gathered through the developed tool described in Section 5.1.1. This section presents the data from this tool for both RC and NRC, starting with the MCC measurements, then files count measurement, followed by Halstead delivered bugs measurement and lastly the LOC change measurements.

McCabe Cyclomatic Complexity Measures The data gathered regarding the MCC is displayed as a chart in Figure 5.1 which was retrieved from the developed tool by using the methods described in Section 2.2.1.1. Looking at the data in the chart, there is a difference between the two components, RC and NRC, not only in the amount of MCC but also in the amount of complex MCC and the distribution between the two components.

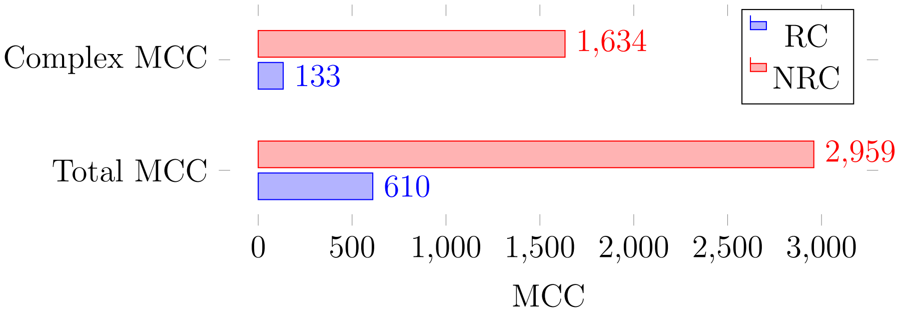

Figure 5.1: Chart displaying the complex MCC and MCC in RC and NRC

The data collected implies that the measure MCC_FOI and MCC_FOI_HIGH are 610 respectively 133 for the RC. This indicates that the MCC_HIGH_% equals to $\frac{133}{610} = 0.2180$ or 21.8% meaning that 21.8% of the MCC found in the RC's FOIs are complex MCC. Although there is some complex MCC, it is not enough to generate any point for the MCC complexity points measure, MCC_POINTS. Zero point means that the measurement does not find any reason to modularize RC's FOIs based on its MCC. Looking at NRC, the same measure MCC_FOI and MCC_FOI_HIGH are 2959 respectively 1634 giving a MCC_HIGH_% of $\frac{1634}{2959} = 0.5522$ or 55.22%. This implies that 55.22% of the MCC are complex in the FOIs of the NRC, which is over the threshold for generating two points to the MCC complexity points, MCC_POINTS. Two points indicates that the measurement system finds the MCC of the NRC's

FOIs to be a problem and, if being developed, has lots of interest attached. The data for RC and NRC are summarized in the Table 5.1.

Measure	RC	NRC
MCC_FOI	610	2959
MCC_FOI_HIGH	133	1634
MCC_HIGH_%	21.80%	55.22%
MCC_POINTS	0	2

Table 5.1: Summary of the MCC data regarding RC and NRC

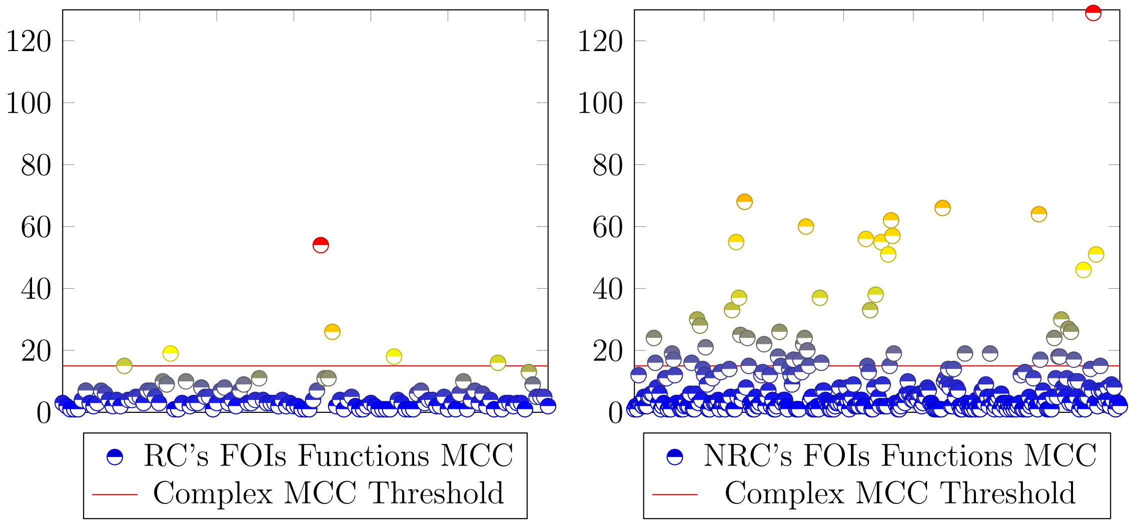

Figure 5.2: A scattered plot displaying the MCC for each function in RC and NRC

Figure 5.2 displays all the MCC values for each function in the two components FOIs. Although the NRC has around 3 times more functions then RC, it still has a higher percentage of functions with complex MCC. Furthermore, the most complex function MCC in NRC is more than twice as large as the most complex function MCC of the RC. Furthermore, it is worth noting that almost half of the complex MCC in RC is generated by one function. While in NRC there are many functions with highly complex MCC.

Files Count Measures The data regarding the number of files in the components as well as the number of FOI are displayed as a graph in Figure 5.3. The tool identified 67 files in the RC and 8 of those files were identified as FOIs. That means that the percentage of FOIs in RC is $\frac{8}{67} = 0.1194$ which is 11.94%. For NRC the tool identified 204 files in total where 17 files were identified as FOIs. That indicates that the percentage of FOIs in NRC are $\frac{17}{204} = 0.0833$ which is 8.33%. As these measures by themselves does not give any useful information, they are used by other measures as a way of normalising. The values are summarized in the Table 5.2.

Halstead Delivered Bugs Measures As mentioned in Section 4.1.2.4 the Halstead is calculated by the sum of all FOIs functions Halstead delivered bugs divided

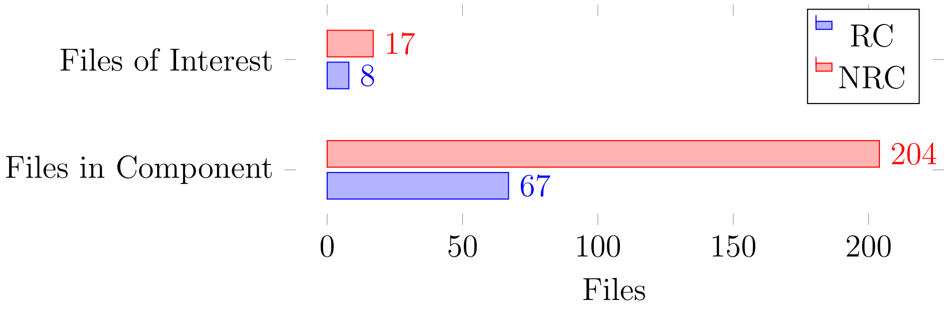

Figure 5.3: Graph displaying the number of files in RC and NRC as well as how many are FOIs

Measure	RC	NRC
FILES_FOI	8	17
FILES_C	67	204
FILES_FOI_%	11.94%	8.33%

Table 5.2: Summary of the MCC measurements regarding RC and NRC

by the number of FOIs. Given the data displayed in Figure 5.4, the sum of RC's FOIs Halstead delivered bugs are 26.4 and for NRC it is 117.23. These values are then normalized by dividing them by the number of FOIs of the respective component (see Section 5.2.1.1) in order to calculate the average number of bugs in the FOIs. For RC this is equals to $\frac{26.4}{8} = 3.3$ and for NRC it is equals to $\frac{117.23}{17} = 6.896 \approx 6.9$. These averages are illustrated in Figure 5.4 as blue lines. The number of points the two components generates are the average Halstead delivered bugs on the FOIs files. The average for RC is 3.3, which is above the threshold for generating 1 point, meaning that the RC component generates 1 point towards the total complexity of FOIs. NRC on the other hand has an average of 6.9 which is above the threshold for generating 2 points. The summarized result for the calculated Halstead delivered bugs can be seen in Table 5.3.

Measure	RC	NRC
HAL_AVG	3.3	6.9
HAL_POINTS	1	2

Table 5.3: Summary of the Halstead delivered bugs measurements regarding RC and NRC.

LOC Change Measures Data regarding the LOC changes are gathered by the tools through the source code change history. The data is visualised in the Figure 5.5 as a chart.

In RC the number of LOC changes in total is 19,949 whereas 1,711 of them are on the FOIs, thus 8.58% of the LOC changes occurred in the FOIs. In NRC the same number are 113,152 and 13,323 respectively implying that 11.77% of the LOC change occurred in the FOIs. Combining these percentages with the percentage of

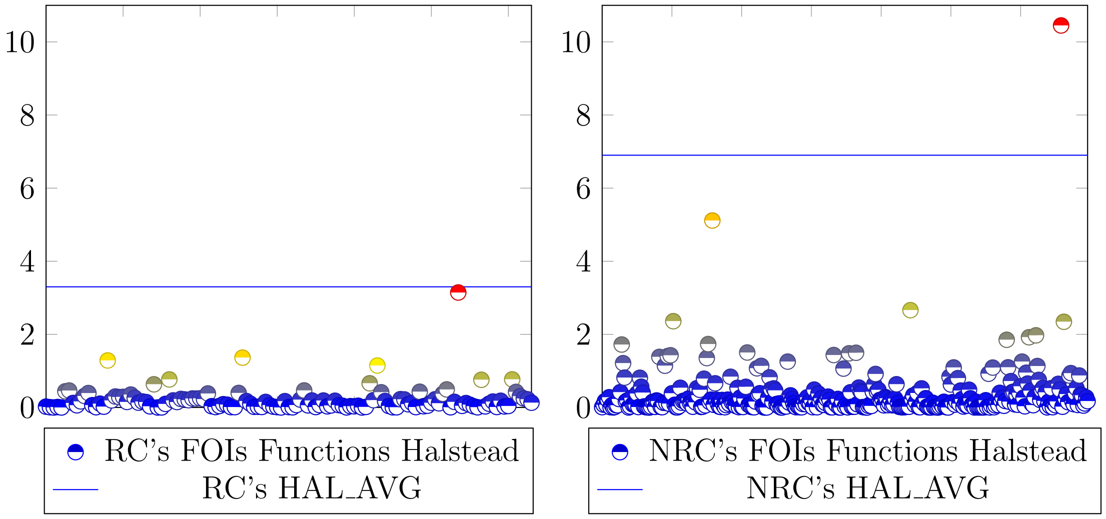

Figure 5.4: A scattered plot displaying the Halstead delivered bugs for each function in RC and NRC. It also displays a blue line for the average Halstead derived bugs for FOIs in RC and NRC.

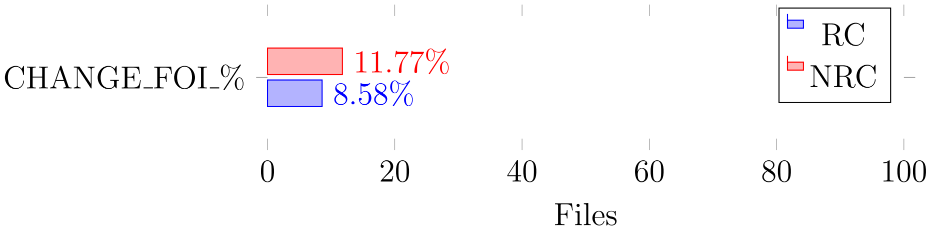

Figure 5.5: Graph displaying the number of LOC changes in RC and NRC as well as how many of them are in FOIs

FOIs in the component (see Section 5.2.1.1) will indicate how likely it is that a FOI will receive a LOC change compared to a non-FOI. For RC this is $\frac{8.58\%}{11.94\%} = 0.7186$ which is 71.86%. NRC is calculate to $\frac{11.77}{8.33\%} = 1.4130$ which is 141.3%. This means that for the RC a FOI is 29.14% less likely to receive a LOC change compared to a non-FOI while for NRC a FOI is 41.3% more likely to receive a LOC change compare to a non-FOI. These measures are summarized in Table 5.4.

Measure	RC	NRC
CHANGE_FOI	1,711	13,323
CHANGE_C	19,949	113,152
CHANGE_FOI_%	8.58%	11.77%
CHANGE_DIFF	71.86%	141.3%

Table 5.4: Summary of the LOC changes measurements regarding RC and NRC

5.2.1.2 Measurement System Result

This section will present the result of the measurement system on RC and NRC by using the data collected in Section 5.2.1.1. This is done by first presenting a table with all the measures that the measurement system calculates, including the not yet presented analysis model and indicator. The analysis model and indicator will then be presented and interpreted, both for RC and NRC.

Measure	RC	NRC
MCC_FOI	610	2959
MCC_FOI_HIGH	133	1634
MCC_HIGH_%	21.80%	55.22%
MCC_POINTS	0	2
FILES_FOI	8	17
FILES_C	67	204
FILES_FOI_%	11.94%	8.33%
HAL_AVG	3.3	6.9
HAL_POINTS	1	2
CHANGE_FOI	1,711	13,323
CHANGE_C	19,949	113,152
CHANGE_FOI_%	8.58%	11.77%
CHANGE_DIFF	71.86%	141.3%
MODULARIZATION_INDEX	0.72	5.65
STATUS	GREEN	RED

Table 5.5: Summary of the all measures and the analysis model of RC and NRC

The summary of all the measurements and the analysis model can be found in Table 5.5. How the different measures has been obtained can be found in Section 5.2.1.1. However, the MODULARIZATION_INDEX and STATUS which are the analysis model and the indicator are not presented in that section, but will be described in this section instead.

Measurement System Result - RC The data from the RC indicates that it is generating one point to the overall complexity of the component's FOIs. The reason for this is that the measurement system does not find enough MCC to indicate that the FOIs have a problem with complexity, but the Halstead delivered bugs measure did indicate the FOIs have a problem. Following the Equation 4.1 to calculate the modularization index, it also uses the FOIs LOC change rate. In the case of the RC this would be 71.86%. That means that the modularization index is calculated to around 0.72 ($\text{MODULARIZATION\_INDEX} = (0 + 1) * 71.86\% \approx 0.72$). Furthermore, looking at the thresholds regarding the STATUS of the measurement system for the RC, it is found that it is set to GREEN. This means that the measurement system did not indicate a need for modularizing the FOIs. However, this does not indicate that the FOIs does not have any complex functions which could justify a refactoring on a file level or function level, but rather the overall complexity of the FOIs does not justify it.

Measurement System Result - NRC The data regarding the NRC does not look as good as it does for the RC, which was expected as the NRC has not been modularized. This is indicated by the fact that it is generating full points regarding the complexity (MCC_POINTS + HAL_POINTS). This means that the measurement system indicates that the implementation of the FOIs were complex in both cases of the complexity measures, MCC and Halstead delivered bugs. Furthermore, as the FOIs LOCs are changed more frequently than the non-FOI this also affect the modularization index further by increasing it. Doing the calculations the modularization index was calculated to 5.65 (MODULARIZATION_INDEX $= (2 + 2) * 141.3\% \approx 5.65$) which is larger than the threshold for the STATUS of RED. This means that the measurement system suggest that the NRC would benefit from being modularized. However, the measurement system cannot determine if the modularization would repay itself. In order to estimate that, further analysis of the FOIs are needed. Such analysis would be the method defined in Section 4.2 and executed for the NRC system in Section 5.2.2.

5.2.2 Effort Estimate Calculations

Given the Equation 4.3 from Section 4.2 this study seeks to use data from both the RC and the NRC to estimate the effort in NRC that can be saved by modularizing the FOIs. This section presents result of the interviews and how the data was applied into the equation. This section begins by presenting the scaling between RC to NRC. After that it begins to present the values for each variable following the method defined in Section 5.1.2.3. Lastly, it will present the calculations of the effort that NRC can save using the Equation 4.3.

5.2.2.1 Problem Size Scaling

As mentioned in Section 5.1.2.1, there is a difference between RC and NRC when it comes to the size of the problem they are solving with the shared functionality. The method that this study will apply is the one described in Section 5.1.2.1 which scales the data regarding the effort from the RC to the NRC.

In order to quantify the problem size the study interviewed the PGs of the two components as well as senior developers with knowledge of both components. This was done for two reasons. The first reason was to collect the data regarding the problem size in a qualitative way. The second reason was to discuss a way from which the problem size could be measured quantitatively.

As for the quantitative way of quantifying the problem size there were a couple of ideas but it would only give a rough estimate. The way that was proposed was to count the number of objects being handled by the in the shared functionality and have the factor between them to be the problem size scale. NRC handled 17 (PROB_SIZE_NRC) of these objects while RC handled 5 (PROB_SIZE_RC). This meant that the problem size difference was a factor of 3.4 (PROB_SCALE) if using this method. However, it was pointed out by the developers and the PGs that this method was a simplified method of defining the size of the problem. The reason for this is that it did not handle the increased difficulty which comes with a new object being added since it was difficult to be obtainable through a quantitative mean.

For the qualitative way, the data was gathered from the interviews where the problem size factor was said to be between 3 to 5 (PROB_SCALE) with an average of 4.5. The result from the qualitative way is almost in similar level as the result from the quantitative in regards to the problem size. With that said, the result from the gathered through the quantitative way will be used throughout the study since it gives an estimation of the problem size.

5.2.2.2 Effort Variables Data

This section presents how the variable data needed for the Equation 4.3 was obtained, following the method described in Section 5.1.2.3.

E_1 is defined as the old development effort of NRC where the data is gathered from the asked questions defined in Section 5.1.2.3. The data received from the PG of NRC was that 100 people are working with the development of NRC; however they do not work full time on the component, but around 20% of their time. Furthermore, the study checked how much effort was spent on FOIs. The study was able to estimate that 9.43% of the effort was spent each month on the development on FOIs files. In order to get the DWM for E_1 the following calculation is done (see Section 5.1.2.2 for details): $E_1 = 100 * 20\% * 9.43\% = 1.886$ DWM/month.

E_2 is defined as the old maintenance effort of the NRC. The number of people working on the maintenance was obtained through the interview questions which are defined in Section 5.1.2.3. The maintenance effort distribution was then gathered from the LOC changes in the source code history for the maintenance. More specific, the data was gathered from the latest maintenance branch in the source control system instead of the main development branch. The latest maintenance branch was chosen over earlier maintenance branches since it would give the most updated information of where the effort was spent in the maintenance. This implies that the effort estimation will be more accurate to the current maintenance compared to if an earlier maintenance branch was to be used. The data received from the PG were that two teams were working full time with the maintenance of NRC which are roughly 12 developers. Additionally 16.94% of all the LOC changes committed in the maintenance branch were in the FOIs. In order to get the DWM for E_2 the following calculation is done: $E_2 = 12 * 16.94\% = 2.0328$ DWM/month.

E_3 defines the estimated effort of refactoring the NRC. Since this has not been done before, the estimation of the refactoring had to be done by appropriate people (see Section 5.1.2.3). These people estimated that it would take a team of seven people around six month to modularize the FOIs of the NRC. This means that that the effort for refactoring would be: $E_3 = 7 * 6 = 42$ DWM.

E_4 defines the development effort of the NRC after the refactoring. It is similar to E_1 but the data could not be extracted from the interview since a refactoring has not been performed. Therefore, the data was gathered from RC and scaled to fit NRC, as described in Section 5.1.2.1. In order to collect the data from RC, the question defined in Section 5.1.2.3 was asked to the people working with RC.

According to the PG of RC there are roughly 40 people working with the RC where they spend 15-20% of their time on the development which gives a mean of 17.5%. The data from the source control system was then used to get the deviation between FOIs and non-FOIs. The result received was that 6.29% of the LOC changes are related to the FOIs. However, the data needs to be scaled in order to fit NRC. Therefore, the method in Section 5.1.2.1 was used, which calculated the scaling to 3.4 in Section 5.2.2.1. Thus, to get the DWM for E_4 the following calculation was done: $E_4 = 40 * 17.5\% * 6.29\% * 3.4 = 1.497$ DWM/month.

E_5 defines the maintenance effort of the NRC after it has been refactored. This is similar to E_2 but is also scaled in the same way as E_4. That means that it uses data from the maintenance branch of the latest release as described for E_2 but also looks at the data of the RC instead of NRC. The question defined in Section 5.1.2.3 for E_5 was asked to the PG of RC to collect the required data. The collected data was that 7 people were working with the maintenance of RC. However, about 15-20% of the time was spent on the maintenance on RC which would give a mean of 17.5%. Furthermore, 3.44% of the LOC changes in the maintenance branch were in FOIs. This was then scaled in order to fit NRC by using the method in Section 5.1.2.1, which calculated the scaling to 3.4 in Section 5.2.2.1. Thus, to get the DWM for E_5 the following calculation was used: $E_5 = 7 * 17.5\% * 3.44\% * 3.4 = 0.143$ DWM/month.

m **and** r defines the amount of time that a release will be maintained. This data was defined in the documents regarding maintenance provided by the company. In this case the maintenance was 18 months long where there is a new release every 6 month. That means that there are three releases being maintained which gives $r = 3$ releases and $m = 6$ months.

Table 5.6 represent the summary of the results of the different variables.

Variable	Value
E_1	1.886 DWM/month
E_2	2.033 DWM/month
E_3	42 DWM
E_4	1.497 DWM/month
E_5	0.143 DWM/month
m	6 month
r	3 releases

Table 5.6: Summary of the variables effort data.

5.2.2.3 Effort Calculation

By using the Equation 4.3 defined in Section 4.2 it is possible to estimate the amount of effort saved after a specific period of time. Using the values summarized in Table 5.6 the equation looks as the following for NRC:

$$
\begin{aligned}
M(x) &= \begin{cases} \frac{r-1}{r}E_2 + \frac{1}{r}E_5 & \text{if } \frac{x}{m} < 1 \\ \frac{r-2}{r}E_2 + \frac{2}{r}E_5 & \text{if } \frac{x}{m} < 2 \\ \vdots & \\ \frac{r-(r-1)}{r}E_2 + \frac{r-1}{r}E_5 & \text{if } \frac{x}{m} < r-1 \\ E_5 & \text{otherwise} \end{cases} \\
&= \begin{cases} \frac{2}{3} * 2.033 + \frac{1}{3} * 0.143 & \text{if } \frac{x}{6} < 1 \\ \frac{1}{3} * 2.033 + \frac{2}{3} * 0.143 & \text{if } \frac{x}{6} < 2 \\ 0.143 & \text{otherwise} \end{cases}
\end{aligned} \tag{5.2}
$$

$$
\begin{aligned}
E &= (E_1 - E_4) * t + \Big(E_2 * t - \sum_{x=1}^{t} \big(M(x)\big)\Big) - E_3 \\
&= (1.886 - 1.497) * t + \Big(2.033 * t - \sum_{x=1}^{t} \big(M(x)\big)\Big) - 42 \\
&= 2.422 * t - \sum_{x=1}^{t} \big(M(x)\big) - 42
\end{aligned}
$$

As shown in the example in Section 4.2.2 it is possible to find out how long after the refactoring that the component will start to save effort. This is done by setting the E variable to zero. This gives the following calculation (assuming $t \geq 12$):

$$
\begin{aligned}
E &= 2.422 * t - \sum_{x=1}^{t} \big(M(x)\big) - 42 \\
&= 2.422 * t - \big((\frac{2}{3} * 2.033 + \frac{1}{3} * 0.143) * 6 \\
&\quad + (\frac{1}{3} * 2.033 + \frac{2}{3} * 0.143) * 6 + 0.143 * (t - 12)\big) - 42 \\
&= 2.279 * t - 53.34 = 0 \\
&\implies t = \frac{53.34}{2.279} = 23.4 \approx 23 \text{ months}
\end{aligned} \tag{5.3}
$$

According to the calculations, part of a component (FOIs) will see a gain in development effort after 23 months. There are a couple of other interesting observations that can be made from the calculation. The total effort on the refactoring, including the double effort of maintaining two solutions for a period of time, is 53.34 DWM. But as mentioned before, 23 month after the refactoring is implemented the developers of NRC will be able to save 2.279 DWM/month. The 2.279 DWM/month is also the interest that is payed each month due to the TD.

This equation is useful when it comes to estimating the effort that can be saved t months after a refactoring. Thereby it answers the sub-research question RQb.

5.2.3 Decision Framework

This section will apply the decision framework described in Section 4.3 on the two components, RC and NRC. Although, the results of applying the different methods are presented in Section 5.2.1 and Section 5.2.2, this section will only refer to them.

RC Follow the decision framework decision graph in Figure 4.5 the first step is to applying the measurement system. The execution of the measurement system on RC can be found in Section 5.2.1, where the indicator was computed to be GREEN. Thus, the measurement system did not find enough complexity in the FOIs together with effort spent on them to warrant a modularization of the FOIs. This can also be interpreted that according to the decision framework, the RC is fine and does not need to be modularized nor does it need to be analysed further.

NRC Same procedure is applied on the NRC, but unlike RC the NRC component is indicated by the measurement system to be RED. This means that the measurement system found that the complexity of the FOIs together with the effort spent on them do warrant for a modularization. However, this alone does not mean that the FOIs should be modularized. The effort required to do the modularization may not get repaid, which would make the refactoring counterproductive. Therefore, the effort estimation calculation needs to be performed. A detailed explanation of how this was done is found in Section 5.2.2. The result of the calculation indicated that it will take 23 months before the total refactoring cost will repay itself, where after that the development effort on the NRC will save around 2 DWM each month. Whether or not it is worth the modularization is up to the developers and is dependent on the how long the component will continued to be developed. Will it continued to be developed, the refactoring can be seen as a long term investment. Also, if the component is to be replaced within the 23 month, the modularization is not an investment.

5.2.4 Validation

As mentioned in Section 5.1.3 two types of validation were conducted. The first validation was conducted with as a survey where developers answered questions regarding the FOIs in RC or NRC about complexity, need of a refactoring and effort. The survey was done for two reasons. One was to compare the results between the developers from each component. The other was to compare the results from the survey with the results from the measurement system, which was hidden from the participant. In total 74 developers were chosen based on the source code change history as well as which teams were working with RC or NRC. Out of these 74 developers 34 were from RC and 40 were from NRC. The reason why NRC had more developers chosen than RC was because there were more developers working with NRC at the time when the study was conducted compared to RC. In total 32 developers answered the survey where 12 were from RC and 20 from NRC. Some of the developers had not worked with the specific functionality that the FOIs represents in the components, but their opinion were still legitimate. Therefore, the study shows the results of the total average value, those who had worked with the FOIs in the components, and those who had not.

The questions which were asked in the survey were:

1. Have you worked with the functionality in RC/NRC? (Yes/No)
2. On a scale of 1-10, where 1 is "Not complex" and 10 is "Very complex" , how complex is it to add/modify/remove source code of the specific functionality in RC/NRC?
3. On a scale of 1-10, where 1 is "Refactoring not needed" and 10 is "Needs refactoring" , how would you rate the need to refactor the specific functionality in RC/NRC?
4. On a scale of 1-10, where 1 is "Not a lot" and 10 is "A lot", how much effort do you think would be saved by refactoring, as mentioned in question 3, in the specific functionality in the RC/NRC?

Validation of RC For RC, 12 developers answered the survey where 8 of them had worked with the FOIs while 4 of them had not. Moreover, Figure 5.6 shows the average total value for both factors, those who had worked with the FOIs in RC, and those who had not. The results for RC indicate that the developers did not find it complex and a refactoring is not needed since it would not save much effort. This is supported from the result in Section 5.2.1.2 where the measurement system indicates that a refactoring of the component is not needed for RC.

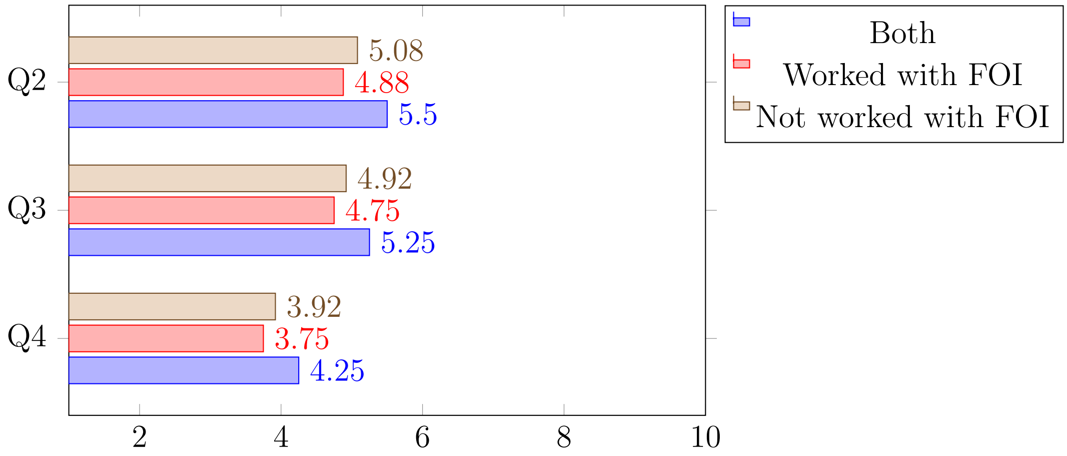

Figure 5.6: Representation of the average value from those who had worked and not worked with the functionality in RC where 1 represent "not a lot" and 10 represents "a lot".

For *Question 2* the developers found that the FOIs was neither "very complex" or "not so complex" when it comes to adding/removing/modifying source code. The total average value regarding *complexity* is 5.08 which is around the normal, while those who had worked with the FOIs got a value of 4.88 and those who had not got a value of 5.5.

For *Question 3* the developers found that a refactoring is not needed but the code could be improved. With that said, the total average value regarding *refactoring* is 4.92 which is around the normal, while those who had worked with the FOIs got a value of 4.75 and those who had not got a value of 5.25.

For *Question 4* the developers found that little effort would be saved after a refactoring and therefore the average value is 3.92 for *effort*. The average value for those who had worked with the FOIs is 4.75 and those who had not is 5.25.

A general observation is that the people who said they had worked with the functionality (FOIs) thought they were simpler than those that had not.

Validation of NRC For NRC, 20 developers answered where 15 of them had worked with the FOIs while 5 of them had not. The Figure 5.7 shows the average total value for both factors, those who had worked with the FOIs in NRC, and those who had not. The results for NRC indicate that the developers found it complex to work with the FOIs in NRC and encourage a refactoring which could save effort. This is supported from the result of the measurement system in Section 5.2.1.2 where it is indicated that a modularization of the component is needed as well as by the effort estimation calculation in Section 5.2.2 where it indicates that there are effort to be saved by doing the refactoring.

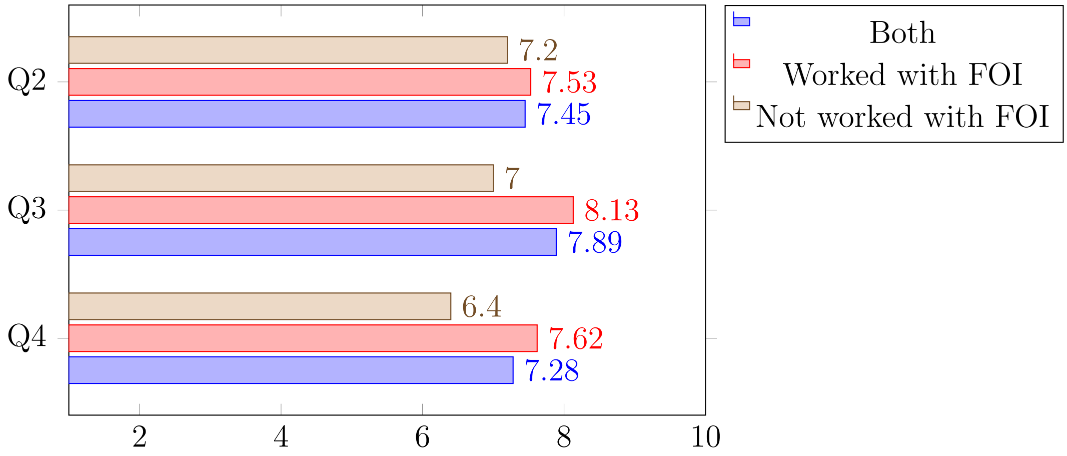

Figure 5.7: Representation of the average value from those who had worked and not worked with the functionality in NRC where 1 represent "not a lot" and 10 represents "a lot".

For *questions 2* the developers found that the FOIs is quite complex to add/modify/remove source code in NRC. The total average value regarding *complexity* is 7.45. Those who had worked with the FOIs got a value of 7.8 and those who had not got a value of 7.2 in NRC. Some comments from the developers who participated on the survey regarding the complexity:

> *"The hard part is to predict how the changes will affect the rest of the code in other areas"*

> *"Hard to get a good grasp of how the different classes work together"*

For *question 3* the developers encourage a refactoring of NRC to simplify the understandability of the code, therefore the total average value is 7.89 for *refactoring.* Those who had worked with the FOIs got an average value of 8.13 while those who had not worked with the FOIs got an average value of 7.

> *"Hard to get a grasp on what's going on there"*

For *question 4* the developers thought that "quite the effort" would be saved if a refactoring would have been performed, thus giving the average total value of 7.28. The developers who had worked with the FOIs believe that more effort will be saved compared to the developers who had not. Those who had worked with the FOIs got a value of 7.62 and those who had not got a value of 6.4 in NRC. Some comments from the developers who participated on the survey regarding the complexity:

> *"Everyone needs to re-learn how everything works again"*
>
> *"More effort would be saved for the maintainers"*
>
> *"The current implementation has a lot of special cases which makes it complex"*

As a general observation, unlike RC were the developers that had not worked with the FOIs though it was more complex than those that had, for NRC it was the opposite.

Interview validation with the PGs As mentioned in Section 5.1.3 an interview validation with the PGs was conducted to get their opinion on how the data was collected and the results of the study. They agreed on how the data was presented and that the threshold used for the measurements system was acceptable. Furthermore, they validated the results of the two methods for the two components, RC and NRC.

RC vs. NRC The results of RC and NRC are quite distinguishable. The results indicate that the developers' find it more complex to work in NRC compared to RC. They also encourage a modularization of NRC compared to the RC which they did not see any large benefits from modularize. This also strengthens the results from the measurement system where it was indicated that RC does not need a refactoring while NRC needs one. To clarify the differences the Figure 5.8 visualizes the comparison between RC and NRC.

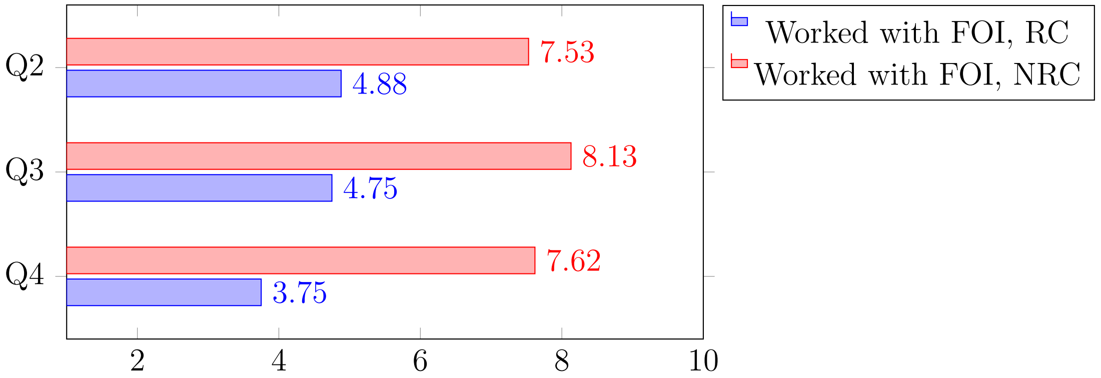

Figure 5.8: Representation of the average value from the survey of those who had worked with the FOIs in RC and NRC where 1 represent "not a lot" and 10 represents "a lot".

Chapter 6

Discussion

This chapter discusses the methods of the study and how they helped to answer the main research question (RQ) and its two sub-research questions, RQa and RQb, asked in Section 1. Firstly, this section will discuss the first method used, the measurement system, which answers the sub-research question RQa. Secondly, there will be a discussion of the second method, the effort estimation calculation, which answers the sub-research question RQb. After that the decision framework which uses the two methods will be discussed and thereby answers the main research question, RQ. The discussion will then move on to the validation followed by: related work; validity and threats; and future work.

6.1 Measurement System

The measurement system allows the stakeholders to be able to analyse different measures which could be used to decide if a component needs to be modularized or not in order to answer RQa. Looking at the different measures regarding a components source code allowed the measurement system to analyse a broader spectrum of the component. For example, this allows the measurement system to suggest that a component does not need to be modularized even though it got a high complexity, which can happened if there has not been many changes to the analysed files then there may not be much interest being paid on those files, even if the debt is high. By trying different combinations of different measures the study was able to find a combination of measures that satisfy the need of the study. These measures were in the areas of complexity and effort and together they were able to analyse the need for a component to be modularized.

6.1.1 Impact on the Industry and the Academia

The design of the measurement system allows it to not only be used in the context of the study, but also, based on the feedback, most likely on other components. The reason for this is that the measures used in the measurement system do not only apply for the context defined in the study. For example, the measurements systems definition what is complex is not limited to the context of this study, but would also be defined as complex in other contexts as well. Applying the measurement system on other components should give the user information regarding if a part of his or her component is in need of being modularized. This is useful for the industry in their decision if to refactoring, but it also has a research focused contribution. Firstly, the measurement system allowed the study to explore the idea of analysing complexity on a specific part of a system with a measurement system, rather than a whole system or specific file. To do this the study took inspiration from other research such as that of Antinyan et al. (2014) and Heitlager et al. (2007) with regards to the measures used. However, the study uses the measures on a different level of abstraction than those papers. Those papers use their measures on a file level, whereas the study uses its measures on a component level. Secondly, it explores measures that are useful when making decisions regarding modularization, more specific the MCC and the Halstead Delivered Bugs. Furthermore, it is worth noting what Pfleeger et al. (1997) mentioned in their paper, a particular model is not guaranteed to be performing well in all circumstances, and there may be some cases where the measurement system does not hold up.

6.1.2 Measures Discussion

In this section the different measures used in the measurement system will be discussed.

6.1.2.1 McCabe Cyclomatic Complexity

The MCC measure was found by the researchers to be a rather good measure on indicating how modularity hides complexity. The reason for this is that modularized components tended to have lower MCC because many of the if- and for-statements for example was move out of the component, and thereby hiding it. Although, it does have some problems such as those pointed out by Hummel (2014), where a function which was perceived by the developer to be easy, due to its structure had a higher MCC than that of a function which was perceived to be more complex.

Even though MCC was originally a measure used on a function level, this study used it in a combination to get the complexity of a group of files. The foundation that was used in order to do this came from Antinyan et al. (2014) where they applied it to file-level. The study found that it scaled well when applied to a component-level as well, where it was well received by the PGs. Similarly, the thresholds used by the measurement system with regard to MCC were elicited from Heitlager et al. (2007) paper. Although, the threshold they used was not meant to be used on a component-level, it did provide a foundation for the thresholds which later were used.

6.1.2.2 Halstead Delivered Bugs

Other studies, such as Alenezi and Abunadi (2015), used LOC and reported bugs to find the defect density. The data that was needed for calculating this could not be gathered automatically by the tool. Instead the study looked a Halstead delivered bugs which could be calculated from the source code. A previous master thesis by Britsman and Tanriverdi (2015) used this measure with a positive result and the PGs agreed that the measure was a good compliment to the MCC in a way of measuring the complexity of a function. The threshold used was based on the knowledge and experience of the researcher as well as discussion with PGs. This means that more research could be done on the thresholds in order to have better threshold values.

As the measure sums the complexity of all the functions in the FOIs, this means that if the FOIs have a lot of LOC compared to their number, they will probably have higher Halstead delivered bugs average. This means that this measure is somewhat effected by the length of the FOIs, which may also be a factor to consider, as longer files are harder to get an overview of.

6.1.2.3 LOC Changes

The overall impression of the LOC changes measure is that they are good measures. The reason for this is that with these measures allowed to estimate how the effort was divided between files. Hence, the amount of effort spent on files are taken into account when deciding if a part of a component needs refactoring. This originates from what McConnell (2011); Guo et al. (2014) wrote regarding the fact that TD retires with the code. Example, a highly complex part of the component may not need refactoring if it is not changed that often. Same goes for the reversed, a slightly complex part of the component that is changed often, should be considered for refactoring.

6.2 Effort Estimation Calculation Discussion

The effort estimation calculation method allows the stakeholders to estimate on how much effort can be saved by modularizing the component after t months, which answers the research question RQb. However, the model also gives information such as how many months it takes until the refactoring will save effort. This is useful information to have when making the decision if to refactoring or not. The reason for this is that if the refactoring will never pay back, then it is not a refactoring worth doing as the effort spent on the refactoring would be more than the effort payed on the interest.

6.2.1 Impact on the Industry and the Academia

The method does have an impact on the industry. As the method allows for estimating the amount of effort that can be saved a certain period of time after the refactoring, it provide useful information to have when making the decision on refactoring. This can be useful for both the developers themselves to know, but also for developers to convince the upper management that refactoring a component is not a waste of effort, but an investment for future development. However, it does not only provide information in support of refactoring. Depending on the case, the model can indicate that refactoring a component will not give a return on the refactoring effort investment. This is just as useful information to have as when the result of the method is that the refactoring investment will give a return.

The method also has impact on the research as well. Previous work, such as Nugroho et al. (2011), investigate the interest of a system or component as a whole and thereby the effect of refactoring the whole system. The method defined in this study investigate the interest on a specific part of a component, the FOIs. Hence, this study contributes to research with an exploratory method of calculating the effort on part of a component, whereas previous studies have calculated the whole component. Furthermore, Sered and Reich (2006) designed an effort calculation on how much the cost would be to design the next generation of component to remove interest. What differentiate this thesis study's method from theirs is that this thesis study focus more on the effort it takes to refactor an existing component to repay TD while their study focus more on the design effort to create a new component to repay TD.

6.2.2 Problem Size Scaling

The problem size scaling was difficult to estimate since there was no clear factor to use when scaling data from RC to NRC. As this is something that is not taken into consideration by other studies, it makes this a first approach in taking this into consideration to scaling data between components. Thus, one way of scaling RC in order to fit NRC was to count the number of objects for each component. Another way was to ask developers who have been working with both components to get an estimation of the problem size. Thus, by not finding any paper discussing on how to decide the scaling (or normalizing) of the problem size, the study decided to go with the number of difference between the objects. With that said, both of these methods are linked to the reliability threat (see Section 6.6.4) because of the estimation made on what the problem size could be.

A more accurate way of scaling the data from between two components depending on the FOI is by scaling dependent on the number of change request affecting the FOIs. This would allow the scaling not to be that dependent on the components to have the same amount of effort ratio put into the FOIs. Although, the most important part when it comes to the scaling is to find a scaling that works for each individual component combination. The reason for this is that for this study the scaling worked the way it is presented in the result, but applying it on another component and the scaling between the problem size may not be as linear.

6.2.3 Effort Estimation Equation

The effort calculation does not take into account the TD that would be accumulated if the component would not be refactored as defined by Martini et al. (2015). This was not considered due to the increase complexity that would be added to the effort calculation. The consequence of this is that doing the refactoring would probably save more effort than the calculation will show. A future work within this area could take this into consideration and thereby making the model more accurate.

6.2.4 Effort Data Collection

There is a limitation in the way the data was elicited. As it stands now it focuses on the historical data. This means that if the equation was to be used to predict the effort saved for future development on FOIs which has been dormant for a long period of time but is going to be active again, then a new way of eliciting the data is needed. This is because the steps described in Section 5.1.2.3 relies on the historical data that may not exist or not to the extent that is needed to give a good estimate.

6.3 Decision Framework Discussion

Using the two methods defined in this study, it is possible to analyse a component to see if it needs to be refactored as well as how much effort could be saved by doing the refactoring. Both these analyses of the component will give the information to a developer or manager to make a good decision regarding if a component needs to be modularized or not. This is because the first method gives the information needed if the component has a problem that would benefit from being modularized, which answers RQa. The second method help with calculating if the effort cost that will be spent on refactoring will be repaid, thereby give the information regarding if the refactoring will help save effort in the future development, which answer RQb. Combining these two methods will provide enough information for making a decision regarding if a component should be refactored or not, which answer RQ.

6.4 Validation Discussion

Validation of Survey During the survey the NRC developers did provide comments regarding their decision. This gave some more insight regarding their opinion of the FOIs. Unfortunately there were not as many comments for the RC.

There was a difference between the answer of the people who had worked with the FOIs and those that had not. For the RC, the developers that had worked with the FOIs though that the FOIs were easier to work with than those who had not worked with them. For the NRC, it was the other way around. The developers that had worked with the FOIs though that they were worse than those that had not.

Connecting the survey result with measurement result The results from the survey was used to connect with the result from the measurement system. As can been seen in Table 5.5 which represents the data from the measurement system while Figure 5.6 and Figure 5.7 represents the data for RC respectively NRC. What can be seen from the measurement system is that there is not a need for a refactoring since RC got a GREEN indication. From the survey, those who have worked with RC had the same opinion as the result from the measurement system. For NRC, the measurement system gave a RED indication which indicates that there needs to be a refactoring. From the survey, those who have worked with NRC had the same opinion and encouraged a refactoring of NRC. Furthermore, those who participated on the NRC survey commented on what the issues were regarding the complexity, need for refactoring and development effort. This was quite valuable since it strengthen the claim the FOIs of NRC being complicated and needs to be modularized.

6.5 Related Work

A paper by Sered and Reich (2006) had an effort estimation method with a different purpose. They investigated what the total cost would be to create a new platform by checking the current cost and the cost of designing next generation of a component. With that said, the paper did not discuss anything about refactoring and TD/ATD.

The paper by Nugroho et al. (2011) developed an approach to quantify TD and its interest. In order to be able to calculate the TD they develop a method which can determine the level of quality of a system. The method was developed to measure and rate the systems quality in terms of the ISO/IEC 9126 standard. It analysis the source code to collect data regarding measures such as LOC, MCC, parameter counts, and dependency counts. A system is given a rating based on the quality level of the system. Using the rating the paper provide equation for calculating the maintenance effort as well as the effort required to upgrade to a new rating by improving the quality level.

Zazworka et al. (2014) did a comparison between four different approaches for identifying technical debt. These techniques were code smell, ASA Issues, Modularity Violations, and Grime. They concluded that the techniques pointed to different problems in the code with a few overlaps of the result. Furthermore, they concluded that not all TD is harmful enough to warrant a repayment specifying the scenario where the cost of paying back the TD outweighs what will be saved in the long term.

Coleman et al. (1994) defined a maintainability index which Heitlager et al. (2007) found a couple of flaws with. For instance, the lack of information of what causes a low maintainability index, which also affects what actions to take to improve the maintainability index. This lead Heitlager et al. (2007) to formulate and apply and alternative model based on source code analysis. To do this they mapped a set of source code measures to the ISO 1926 standard. Examples of measures they chose to use were MCC, code duplication, LOC, and test coverage. To these measures they define thresholds ranking five levels, from good to bad.

Another study by Antinyan et al. (2014) developed a method and a supporting tool, in the form of a measurement system, to use in Agile/Lean production for identifying and assessing risk of deploying new code. The method was based on two properties: complexity and revision of a file. The complexity method they chose to use was that of MCC, although to be able to distinguish between files with many small non-complex functions and files with a few large complex functions the study defined something they called *effective cyclomatic complexity percentage of a file*. Using this together with the number of revision on a file, the researchers were able to identify files which was deemed by developers to be risk with 95% accuracy.

6.6 Validity Threats

In this section the validity threats will be discussed based on the four types of validity outlined by Runeson and Höst (2009): internal validity, external validity, construct validity, and reliability validity.

6.6.1 Internal Validity

Internal validity is when a third factor can affect the study (Runeson and Höst, 2009). Scaling the data from another component using difference in the problem size has a couple of flaws. The reason for this is that the problem size is a third factor that is not controlled by the study. First off, if the FOIs are in the component that is used to scale does not change with the same frequency as the analysed component, then the scaling may be off and affect the result. Secondly, the effort might not change linear to the problem scaling, which will affect the result. To minimize the effect of this threat the study validated the result with software experts at the company.

The effort data was gathered though interviews and estimations. This means the effort could be caused by something other than the TD that was asked. Meaning that there might be other factor that makes the interviewee give false data, such as current problems they are facing or thinking the results will be used to measure their performance. To avoid this, the study triangulated the data by interviewing both developer from different teams and the PGs.

6.6.2 External Validity

The external validity concerns to what extent it is possible to generalize the findings, and also to what degree the finding are interesting to other departments within the organization (Runeson and Höst, 2009). The study was conducted at one company where two components were analysed. The decision framework has not been tested for another component, another department or company, and therefore difficult to say if it is applicable outside said company.

The study is measuring the complexity for C++ files. Thus, if the study were to measure the complexity of the files in another language, for example Java, the number of complexity might be different. This means that the threshold that is used by the measurement system may only apply on the C++/C.

In the effort estimation calculation it is assumed that all maintained releases get an equal share of the maintenance effort. This means that if a component maintenance situation is different, when it comes to the distribution of the effort spent on previous release, the effort estimation calculation needs to be adjusted to account for these discrepancies. For example, if 2 releases are maintained and the latest of them receive 80% of the effort spent on maintenance, then the calculation model needs to be adapted as it assumes that both the releases receive 50% each of the effort spent on maintenance.

6.6.3 Construct Validity

Construct validity concerns what the study is trying to investigate in order to answer the research questions. When it comes to the FOIs, it may be hard to distinguish FOIs and non-FOI. This is because some files may have functionality that belongs to a FOI as well as functionality that do not. Hence, this will give a conflict within files and put a threat to the result of both the measurement system and the effort estimation calculations defined in this study. For this study the FOIs were chosen only if its functionality were that of a FOI. This allowed the result of the thesis to be pessimistic. Not only that, the study analysed both the source code as well as interview experts on the different component to make sure correct files were analysed.

Most of the implementation regarding the functionality analysed are in the identified FOIs. However, some non-FOIs files may still have some implementation that is associated with functionality, yet they are not counted towards the overall complexity of the decision framework developed in this study. The reason for this is that if these file would be identified as FOIs, effort and complexity not related to the functionality analysed would contaminated the FOI data.

The measurement system does not take other measurements such as code duplication and number of dependencies into consideration. Therefore, complexity may exist in a component that the measurement system is indicating as GREEN. The reason for this is that the measures of the measurement system cannot detect that type of complexity, and thereby they will be hidden and affect the result. However, the study did validate the measurement system to make sure that it found the complexity it was designed to find.

During the interview the participants had to be introduced to the terminology and also the purpose of the interview so they did not misunderstood the questions, and thereby give false data. A survey was also created to validate with the developers of RC and NRC regarding their opinion about the complexity, the need for refactoring and effort of the FOIs in RC respectively NRC. The construct validity of the survey was that the questions were sent via email to the participants. The participants could misunderstand and interpret the questions and thereby give false data. To minimize the risk of this happening the study used a pilot study on the survey.

The measures regarding the LOC changes are gathered through the source control system. Although, an optimal solution would be to analyse all the LOC changes in each of the file to find out more information about them. For example, how many LOC changes are empty lines, or even how many LOC where changed/deleted/added. The developed tool that was created to get the data only collects the number of row deleted and/or added. Moreover, changing one line of code will in fact generate two LOC changes in the system. This is because the changed row is counted by the source control system to have been deleted and then added again. These were things the authors did take into action due to low resources in terms of developers. Although the data calculated with this data was validated with people responsible for the components.

The tool, which was used to collect the data for the measurement system was developed by the study's authors. The tool itself may have some bugs which the authors have not detected. However, each function of the tool was tested on multiple files of different length and content to make sure that the outcome was correct.

6.6.4 Reliability

Reliability threat concerns on how reliable the results of the study is. For example, would other researches produce the same output by conducting the same study? A reliability threat for this study was collecting the data of RC and NRC for the effort method. Due to the lack of documentation of the data, regarding on how many people are working with each component and how much time they spend on the development and maintenance, interviews had to be conducted with participants from RC and NRC as well as PGs. If other researchers were to conduct the same procedure they could potentially get different values due to the lack of documentation as mentioned previously. The reason for this difference is that the interviewees' experience and perception of the FOIs may change. To avoid this the study triangulated the data by interviewing different roles as well as people from multiple team.

6.7 Future Work

This study explored different measures that can be used to decide if a part of a component needs to be modularized. There are still measures that the study did not consider, or simply did not have time to explore. This means that there are room for further exploration which is not only limited to new measures, but also on the thresholds used by the measurement system. Although, the thresholds were good enough for the components analysed in this study, a future study can do a deeper analysis to make sure they are acceptable for other components as well.

When it comes to the effort estimation calculation there are a couple of things that can be developed further. Firstly, the equation does not take into account the increased interest that is added as future development increases the TD of the component. Secondly, this study only calculates on the effort saved by refactoring. However, one could further develop the method to take into account the estimated total financial cost of refactoring.

Chapter 7

Conclusion

Architectural Technical Debt (ATD) is known to be a problem when handling software project nowadays. Refactoring a component to remove the debt is a major decision as there are many factors that need to be considered. As time goes, unmanaged debt will increase for the component which will delay new feature release since more time is spent on the development and maintenance.

This study aims to help the developers making an informed decision regarding if to modularize part of a component into its own component or framework. This was done by a decision framework which uses the two developed methods. These methods provide information that are valuable when it comes to make an informed decision regarding the modularization. The first method is a measurement system which analysed the source code statically to check if the analysed part of the component's source code contains enough TD combined with the effort spent to warrant a modularization. The second method is an effort estimation calculation that is to be used to estimate the impact on the effort a refactoring would have on the continued development of the component. This will help developers as well as managers to make a more informed decision.

The key findings of this study are:

- *A measurement system* which is based on the ISO 15939 standard to analyse source code statically to find if it would benefit from being modularized. The measures that are used in the measurement system are used in order to conclude if these are complexity measures and effort measures. The measurement system was developed to answer sub-research question RQa.

- *An effort estimation calculation* that can be used to estimate the effort saved by repaying TD though a refactoring. To do this, it uses the data from the analysed component regarding the current development effort as well as the current maintenance effort. The calculation also takes into account the double maintenance that occurs when multiple releases are maintained. This answer the sub-research question RQb. Furthermore, the equation used by the calculation can also be used to find out when the refactoring will have repaid itself.

- *A Decision Framework* which helps developers to decide if they would benefit from modularizing a specific part of a component. It uses the two methods mentioned previously in its decision making. This answers the main research question RQ.

In summary, the findings of this study will help in the decision making if to refactor part of a component by modularizing it. The decision framework uses the measurement system to decide if the component would benefit from being modularized. It will also use the effort estimation calculation to get an estimation if the modularization will repay itself and how much effort will be earned afterwards. By using this decision framework a developer will have a foundation when they are making the decision to modularize part of a component as well as a prediction of the benefits in the form of effort.

Bibliography

S. Hassaine, "Evaluating design decay during software evolution," p. 159, 2012, copyright - Copyright ProQuest, UMI Dissertations Publishing 2012; Last updated - 2015-08-28; First page - n/a. [Online]. Available: http://search.proquest.com/docview/1353181128?accountid=10041

W. Cunningham, "The wycash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://dl.acm.org/citation.cfm?doid=157709.157715

P. Kruchten, R. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 51–54, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2507326

R. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, Aug 2012, pp. 91–100.

Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 31–34. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1985362.1985370

A. Martini, J. Bosch, and M. Chaudron, "Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study," *Information and Software Technology*, vol. 67, pp. 237 – 253, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584915001287

I. Gat and J. D. Heintz, "From assessment to reduction: How cutter consortium helps rein in millions of dollars in technical debt," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, ser. MTD '11. New York, NY, USA: ACM, 2011, pp. 24–26. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1985362.1985368

J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the*

2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1852786.1852797

V. Antinyan, M. Staron, W. Meding, P. Osterstrom, E. Wikstrom, J. Wranker, A. Henriksson, and J. Hansson, "Identifying risky areas of software code in agile/lean software development: An industrial experience report," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on.* IEEE, 2014, pp. 154–163.

I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the.* IEEE, 2007, pp. 30–39.

A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2nd Workshop on Managing Technical Debt.* ACM, 2011, pp. 1–8.

A. Martini, J. Bosch, and M. Chaudron, "Architecture technical debt: Understanding causes and a qualitative model," in *40th Euromicro Conference on Software Engineering and Advanced Applications*, 2014, pp. 85–92.

N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 09 2014. [Online]. Available: http://search.proquest.com/docview/1540413116?accountid=10041

M. Fowler, http://martinfowler.com/bliki/TechnicalDebt.html, 2003.

Y. Guo, R. O. Spínola, and C. Seaman, "Exploring the costs of technical debt management – a case study," *Empirical Software Engineering*, 2014. [Online]. Available: http://link.springer.com/article/10.1007%2Fs10664-014-9351-7

S. McConnell, "Managing technical debt webinar," 2011, accessed: Sep 2015. [Online]. Available: https://www.youtube.com/watch?v=lEKvzEyNtbk

T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

T. Yin, http://www.lizard.ws/, 2015.

KRIS, http://asetechs.com/NewSite2015/Products/Interpreting_Halstead_metrics.htm, 2001.

Verifysoft, http://www.verifysoft.com/en_halstead_metrics.html, 2010.

D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, “Investigating the impact of design debt on software quality,” in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, ser. MTD ’11. New York, NY, USA: ACM, 2011, pp. 17–23. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1985362.1985366

N. Zazworka, R. O. Spínola, A. Vetro’, F. Shull, and C. Seaman, “A case study on effectively identifying technical debt,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE ’13. New York, NY, USA: ACM, 2013, pp. 42–47. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2460999.2461005

Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” *Journal of Systems and Software*, vol. 101, pp. 193 – 220, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214002854

A. Martini and J. Bosch, “The danger of architectural technical debt: Contagious debt and vicious circles,” in *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, May 2015, pp. 1–10.

S. Wong, Y. Cai, M. Kim, and M. Dalton, “Detecting software modularity violations,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 411–420. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1985793.1985850

“Systems and software engineering – vocabulary,” *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò, “Using technical debt data in decision making: Potential decision approaches,” in *Proceedings of the Third International Workshop on Managing Technical Debt*, ser. MTD ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 45–48. [Online]. Available: http://dl.acm.org.proxy.lib.chalmers.se/citation.cfm?id=2666036.2666044

P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

S. Hove and B. Anda, “Experiences from conducting semi-structured interviews in empirical software engineering research,” in *Software Metrics, 2005. 11th IEEE International Symposium*, Sept 2005, pp. 10 pp.–23.

J. Linaker, S. M. Sulaman, R. Maiani de Mello, M. Hˆst, and P. Runeson, “Guidelines for conducting surveys in software engineering,” 2015.

IEEE, “Ieee standard adoption of iso/iec 15939:2007 systems and software engineering measurement process,” *IEEE Std 15939-2008*, pp. C1–40, Jan 2009.

M. Staron, W. Meding, and C. Nilsson, “A framework for developing measurement systems and its industrial evaluation,” *Information and Software Technology*, vol. 51, no. 4, pp. 721–737, 2009.

B. Hummel. (2014) Mccabe’s cyclomatic complexity and why we don’t use it. [Online]. Available: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/

C. Jones, “Software metrics: good, bad and missing,” *Computer*, vol. 27, no. 9, pp. 98–100, Sept 1994.

S. L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham, “Status report on software measurement,” *IEEE software*, no. 2, pp. 33–43, 1997.

M. Alenezi and I. Abunadi, “Quality of open source systems from product metrics perspective,” *International Journal of Computer Science Issues (IJCSI)*, vol. 12, no. 5, pp. 143–148, 09 2015. [Online]. Available: http://search.proquest.com/docview/1729353283?accountid=10041

E. Britsman and Ö. Tanriverdi, “Identifying technical debt impact on maintenance effort-an industrial case study,” 2015. [Online]. Available: http://hdl.handle.net/2077/40110

Y. Sered and Y. Reich, “Standardization and modularization driven by minimizing overall process effort,” *Computer-Aided Design*, vol. 38, no. 5, pp. 405 – 416, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010448505001910