HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Software Business and Engineering Institute

Mika Mäntylä

Two Experiments on Subjective Evaluation of Code Evolvability

Licentiate thesis submitted for official examination for the degree of Licentiate in Technology.

Espoo, 8.12.2005

Supervisor: Tomi Männistö

Instructor: Tomi Männistö

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF LICENTIATE THESIS**
Department of Computer Science and Engineering

Author
Mika Mäntylä

Date
8.12.2005

Pages
75

Title of thesis
Two Experiments on Subjective Evaluation of Code Evolvability

Professorship
Software Engineering

Professorship Code
T-76

Supervisor
Tomi Männistö

Instructor
Tomi Männistö

Software evolvability – the ease of further developing software – is an important quality attribute greatly dictating the future potential of any software system. Recent trends such as agile software development and extreme programming have highlighted refactoring – modifying the internal structure of software without affecting its observable behaviour – as a key factor for ensuring software evolvability. To help software developers in making a refactoring decision, descriptions of bad code, so called *code smells* have been proposed. Although humans play an important role in making refactoring decisions, most of the work around refactoring has focused on tools and metrics, instead of empirical work with humans. Therefore, we studied refactoring decisions and the evaluation of the existence of code smells at the source code method level.

Two experiments were made with a different sample of students in each. The participants evaluated the existence of certain code smells for 10 methods, stated whether each method should be refactored or not, and provided rationales for their refactoring decisions. We studied the interrater agreement, i.e. the extent to which evaluators agree. Furthermore, we studied the explaining factors of the evaluations with three approaches. First, with regression analysis we studied how the measures of the evaluated code and the background of the evaluators affected the evaluations. Second, we analyzed the information of the refactoring decision rationales. Third, we used the information discovered in the rationales to explain the refactoring decision with regression analysis.

The results showed high interrater agreement for the simple code smells, and low agreement for the more complex smell and the refactoring decision. The code metrics explained over 70% of the variation regarding the simple code smell evaluations, but only about 30% of the refactoring decision. Surprisingly, the demographics were not useful predictors neither for evaluating the code smells nor the refactoring decision. The best predictors of the refactoring decision were the rationales of the refactoring decision, i.e. the qualitative data, explaining over 70% of the variation, and the evaluations of the code smells explaining more than 60% of the variation. The analysis of qualitative data showed that the method under study affects the contents of the rationales. In other words with different methods we would have obtained different rationales. The qualitative data indicates that automatic detection of some code problems would require new code metrics, and some problems could not be detected at all.

The results suggest code metrics tools usage as an effective approach in highlighting straightforward problems in the code. The low interrater agreement of the refactoring decisions, the poorly performing code metric based regression models of the refactoring decisions, and the fact that some refactoring decision rationales are difficult or impossible to measure, indicate difficulty in building tool support simulating real-life subjective refactoring decisions. The main outcome of this study is that commonly used code metrics alone are not sufficient predictors for refactoring decisions and that qualitative data offers far superior insight of refactoring decisions. Therefore, we need to make further studies on the qualitative elements of software evolvability. Hopefully, with qualitative studies we can discover the true evolvability problems in code, define new code metrics, and, thus, enable creation of effective tool support simulating real-life subjective refactoring decisions.

Keywords:
software engineering experiment, software evolvability, software maintainability, subjective evaluation, interrater agreement, regression analysis, qualitative analysis, coding paradigm

TEKNILLINEN KORKEAKOULU Tietotekniikan osasto	**LISENSIAATINTUTKIMUKSEN TIIVISTELMÄ**
Tekijä Mika Mäntylä	Päiväys 8.12.2005
	Sivumäärä 75
Työn nimi Kaksi koetta ohjelmiston jatkokehitettävyyden subjektiivisesta arvioinnista	
Professuuri Ohjelmistotuotanto	Professuurin koodi T-76
Työn valvoja Tomi Männistö	
Työn ohjaaja Tomi Männistö	

Ohjelmiston jatkokehitettävyys on tärkeä laatuominaisuus, joka vaikuttaa ratkaisevasti ohjelmistojärjestelmän tulevaisuuden mahdollisuuksiin. Erityisesti ketterä ohjelmistokehitys korostaa refaktoroinnin – ohjelmiston rakenteen muokkaamisen muuttamatta sen ulkoista toimintaa – tärkeyttä ohjelmiston jatkokehitettävyyden varmistamisessa. *Koodihajuja*, jotka ovat kuvauksia huonosta ohjelmakoodista, on esitetty ohjelmistokehittäjien avuksi heidän päättäessään ohjelmakoodin refaktoroinnista. Ihmiset ovat avainasemassa refaktorointipäätöksissä, mutta alueen aiempi tutkimus on silti keskittynyt pääasiassa työkaluihin ja koodimittareihin eikä empiiristä tutkimusta ihmisten roolista ole juurikaan tehty. Työssä tutkittiin ihmisten refaktorointipäätöksiä ja arvioita koodihajujen esiintymisestä ohjelman metoditasolla.

Työssä tehtiin kaksi koetta kahdella eri otoksella opiskelijapopulaatiosta. Koskien kymmentä metodia osallistujat arvioivat koodihajujen esiintymistä ja refaktoroinnin tarvetta sekä perustelivat refaktorointipäätöksensä. Kokeen tuloksista analysointiin arvioijien keskinäistä samanmielisyyttä. Lisäksi selvitimme kolmella eri tavalla arvioinnin tulokseen vaikuttuvia tekijöitä. Ensin tutkimme regressioanalyysillä kuinka hyvin mittaukset arvioidusta koodista ja arvioijan taustatiedot pystyivät selittämään refaktorointipäätöstä. Seuraavaksi tutkimme mitä asioita laadullinen aineisto, eli refaktorointipäätösten perustelut, sisälsi. Lopuksi tutkimme kuinka hyvin refaktorointipäätösten perusteluista löytyneet eri asiat pystyivät selittämään refaktorointipäätöstä.

Tulokset osoittivat, että arvioiden keskinäinen yhdenmielisyys oli korkeaa yksinkertaisissa koodihajuissa ja matalaa kompleksisissa koodihajuissa ja refaktorointipäätöksissä. Koodimittaripohjaiset regressiomallit selittivät yli 70% yksinkertaisten koodihajujen arviosta, mutta vain 30% refaktorointipäätöksistä. Yllättäen arvioitsijoiden taustatiedot eivät vaikuttaneet koodihajuarvioihin tai refaktorointipäätöksiin. Parhaiten refaktorointipäätöstä selittivät refaktorointipäätöksen perustelut kattaen yli 70% variaatiosta ja koodihajuarviot kattaen yli 60% variaatiosta. Analysoidessamme refaktorointipäätösten perusteluita huomasimme, että arvioitava metodi vaikuttaa suuresti perusteluiden sisältöön. Eli erilaisilla arvioitavilla metodeilla olisimme saaneet huomattavasti toisenlaisia perusteluita. Perusteluiden analyysi myös osoitti, että löytääksemme automaattisesti osan koodiongelmista niille pitäisi luoda uusia lähdekoodimetriikoita ja osaa koodiongelmista ei voitaisi löytää koodimittareilla lainkaan.

Tulosten perusteella näyttää siltä, että koodimittarityökalun käytöllä voidaan tehokkaasti löytää suoraviivaisia ongelmia koodista. Refaktorointipäätösten matala arvioijien välinen yhdenmielisyys ja heikosti menestyneet koodimittaripohjaiset regressiomallit, sekä mahdottomuus mitata kaikkia refaktorointipäätöksen perusteluita saattavat estää tosielämän subjektiivisia refaktorointipäätöksiä simuloivan työkalun toimivuuden. Tutkimuksen päätulos on, että yleisesti käytetyt koodimetriikat eivät yksinään ole riittävän hyviä ennustamaan refaktorointipäätöstä. Kvalitatiivinen aineisto mahdollistaa paremman refaktorointipäätösten ennustamisen. Näin ollen on tarve jatkotutkimukselle, joka keskittyy ohjelmiston jatkokehitettävyyden tutkimiseen laadullisten menetelmien kautta. Laadullisen tutkimuksen avulla voimme tunnistaa todellisia ongelmia koodissa, kehittää uusia koodimittareita ja näin ollen mahdollistaa työkalutuen, joka simuloi tosielämän refaktorointipäätöksiä.

Avainsanat:
ohjelmistotuotannon koe, ohjelmiston jatkokehitettävyys, ohjelmiston ylläpidettävyys, subjektiivinen arviointi, vastaajien välinen yhdenmielisyys, regressioanalyysi, laadullinen analyysi

Acknowledgements

During the thesis work, I have had the pleasure to work with many inspiring individuals whose ideas, support and friendship have greatly improved the outcome of this work.

Juha Itkonen provided ideas in planning the experiments and helped with the practical arrangements. Pietu Pohjalainen and Mikko Rusama piloted the first experiment and gave valuable feed back. Jari Vanhanen and Kristian Rautiainen reviewed parts of this work and helped in the experiment planning. Discussions with my original supervisor and current research group leader Casper Lassenius have greatly contributed to this work. My supervisor and instructor Tomi Männistö gave me valuable ideas and helped me in finalizing this work. Jarno Vähäniitty handled the tasks of a project manager in our research project (SHAPE) and provided many moments of laughter and joy. In addition, all other members of our research group (SPRG) need be acknowledged for their support. In the end, the experiment could not have taken place without the students of the Software Testing and Quality Assurance course who participated in the experiments.

I also want to thank my mother Sinikka and my father Ilmari for their support. Finally, I thank my spouse Terhi for the love, companionship and support she has given me.

Espoo 8.12.2005

Mika Mäntylä

Table of Contents

1 Introduction

1.1 Motivation and Background

Software evolvability – the ease of further developing software – is an important quality attribute greatly dictating the future potential of any software system. In the past, there was a strong emphasis on up front design for ensuring software evolvability. However, recent trends such as agile software development and extreme programming have highlighted refactoring – modifying the internal structure of software without affecting its observable behaviour – as a key factor for ensuring software evolvability. For example, Microsoft has recognized the constant need to modify existing software structure to ease future development. Therefore, Microsoft's Office division determined that 20% of development effort should be budgeted to code modification (Cusumano and Selby 1995 pp. 280-281).

An important issue concerning software evolvability is the decision when to perform refactoring. It seems likely that wrong refactoring decisions can do more harm than good. Fowler and Beck have come up with a term called *code smell* (Fowler and Beck 2000) to help software developers in recognizing problematic code. These code smells are general descriptions of bad code that are supposed to help software developers decide when the code needs refactoring. Fowler and Beck (2000) claim that exact criteria for refactoring decisions cannot be given: *"no set of metrics rivals informed human intuition"*.

Thus, humans play an important role in making software-refactoring decisions. Still, most of the work around refactoring has focused on tools and metrics, see (Mens and Tourwe 2004) for details. There are a limited number of empirical studies and controlled experiments studying subjective software evolvability evaluation, i.e. refactoring decisions and the evaluation of the existence of code smells. We studied this topic at the source code method level. Two experiments were made with a different set of students in each. The participants evaluated the existence of certain code smells for each method and then stated whether the method should be refactored or not. In the second experiment, the participants additionally stated the rationale for the refactoring decision. Our first objective was to assess the interrater agreement, i.e. the extent to which evaluators agree. High interrater agreement is a positive indication of the reliability of the subjective evaluations. Lack of interrater agreement can mean that some evaluators are mistaken in their evaluations. The second objective was to study how factors, such as the evaluated code itself and the background of the evaluators, affect the evaluations. An analysis of these factors can help us find predictors for the code smell evaluations and the refactoring decisions, which can be used, e.g. in building tool support. The third objective was to study the qualitative reasons for refactoring, i.e. the refactoring decision rationales. From the qualitative information, we can perhaps exact criteria to help developers in creating more evolvable code. Qualitative information can also help in creating new metrics that could measure the refactoring need.

This work builds on our previous study (Mäntylä et al. 2003; Mäntylä 2003; Mäntylä et al. 2004) of subjective evaluation of software evolvability. This study tries to fix many of the shortcomings of our previous studies, particularly those that were concerned with the statistical power, and the reliability of the procedures performed by informants.

1.2 Terminology

In this work, software evolvability stands for *the ease of further developing a software element*. This term was chosen over more traditional term software maintainability for the reasons covered in Section 2.1.

Subjective evaluation or *perceived evaluation* means an evaluation performed by an individual. In real world, such evaluations are performed for example by ski jumping, beauty contestant, and wine tasting judges. In context of this work, subjective evaluation is performed on source code.

1.3 Research Problem and Scope

The objective of this research is to study empirically the subjective evaluations of software evolvability. The research problem is:

Do human evaluations of software evolvability differ from one another, and what are the explaining factors behind the evolvability evaluations?

The research problem is answered by studying the following research questions, which are further elaborated in Section 3.1

Research question 1: *Is there an interrater agreement in subjective evolvability evaluations?*

Research question 2: *How much of the evolvability evaluation of a software element can be explained by the software element and the informant?*

Research question 3a: *What factors act as the rationales for the refactoring decision, could these factors be automatically detected, and what would be the effect of the improvement suggestion factors to common source code measures?*

Research question 3b: *Can the factors of the rationales predict the refactoring decision, and, if yes what are the most important factors?*

The data needed to answer the research questions is provided by two experiments. Details of the experiments are in Section 3.2. These research questions are studied within following scope

Evolvability evaluation, including both the evaluation of bad elements in the code and the decision whether to improve the code by refactoring, of software methods based principally on the software structure using human evaluations.

The scope is discussed in Section 2.3, which also provides on overview of the research area demonstrating the areas outside of the scope of this work.

1.4 Structure and Outline of the Thesis

Rest of the thesis is organized as follows. Chapter 2 discusses different viewpoints of software evolvability, present the scope of the work, and introduce the relevant prior work. Chapter 3 elaborates the research questions and the methodology utilized to provide answers to the research questions. Chapter 4 presents he results of that are used to answer the research question. Chapter 5 answers the research questions, asses the limitations of the study. Finally, Chapter 6 presents the conclusion of the study and provides direction for future work.

2 Background

This chapter positions the work by introducing the term software evolvability, and highlighting different viewpoints to software evolvability and to software evolvability evaluation. The scope of the work is presented at end of Section 2.3. Relevant prior work is summarized in Section 2.4. Finally, Section 2.5 points out the gaps in existing literature.

2.1 Software Evolvability

In this study, we use the term *software evolvability* to denote *"the ease of further developing a software element"*. Traditionally, the term *software maintainability* has been used to represent this quality attribute, and IEEE (IEEE 1990) has defined software maintainability as follows: *The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment.* Additionally, Pigoski (Pigoski 1996) quotes several sources that contain almost similar definitions for the term software maintainability.

Our definition of software evolvability is more restrictive than the definition of software maintainability, which typically includes the ease of fault correction (corrective maintenance) and adaptation (adaptive maintenance). The term software evolvability has a close match with the term *perfective software maintenance*, which according to the IEEE standard glossary of software engineering terminology (IEEE 1990) is defined as: *software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.*

Historically, software maintainability has a strong link to the maintenance phase of the software lifecycle. However, the term software maintenance poorly describes what typically happens after initial software deployment. The word *maintainability* is derived from the verb "maintain", which according to Merriam-Webster's dictionary[1] can be defined as: "*To keep in an existing state (as of repair, efficiency, or validity)*; *To preserve from failure or decline (maintain machinery).*" Thus, the word maintainability refers to our ability to keep software in an existing state or to preserve it from decline. The problem with this definition is that software is not consumed or worn down by use. However, most software systems, and software products in particular, are subject to lots of changes after their initial deployment. A large part of these changes are extensions to the existing system (sometimes referred to as perfective maintenance). Whereas the term maintenance may have been representative and useful in the 1970s, it fits poorly with modern iterative development processes and the constant evolution of contemporary software systems.

We think that the term *software evolution* better describes what happens after the initial software deployment/release, an idea also supported by Sommerville (Sommerville 2001). Rajlich and Bennett (Rajlich and Bennett 2000) propose an improvement to the traditional develop and maintain model by presenting a life cycle model that also describes the phases after the software has been released. In Rajlich's and Bennett's model, software evolution is also seen as an important phase in the software lifecycle.

[1] http://www.m-w.com/

The term software maintenance also offers a poor match with the development and release of software products. In software system development, where typically a custom-made system is delivered to a single client, the maintenance phase can be clearly identified after the software has been delivered to the customer. In software product development, the development is evolutionary and there are several deliveries to different customers. In the software product business there is a constant need to further develop the product because of the continuous competition for customers, i.e., a company will add new features to their product to attract new customers. Therefore, we do not think that the term software maintenance is appropriate in the software product context to describe all the modifications made to a software product after it has been initially shipped.

Since using the term software maintenance or the verb maintain to refer to the modifications made to the software offer a poor match with the real world phenomena, we have chosen to use the term software evolvability rather than the traditional term software maintainability to describe the ease of developing a software element further. The term software evolvability could be substituted with perfective software maintainability, but for the reasons listed in this section, we have opted not to do this.

2.2 Viewpoints to Software Evolvability

We have identified four viewpoints to software evolvability, as shown in Table 1. Firstly, we may study factors that affect software evolvability, e.g., why a piece of software has become poorly evolvable. The list of those factors is likely to be extensive, covering issues from the programming language used and the motivation of the developers to the business goals and the organization of the developing company. Some work in this area has been done by Oman et al. (Oman et al. 1991)[2], who listed different factors affecting software evolvability. Lehman (Lehman 1980) has proposed laws that affect software evolution, some of which also affect software evolvability.

Secondly, we can look at how evolvable a piece of software is at the moment. Evolvability can be evaluated by looking at the software element itself, as well as its documentation. Software evolvability is likely to be dependent on the evaluator. For example, the evolvability of a software element can be high to the original developer, but at the same time, a new developer who lacks proper knowledge of e.g. the used development paradigm can experience great difficulties. Another way to evaluate the evolvability is through automatic program analysis. We discuss evolvability evaluation in more detail in Section 2.3.

Thirdly, we can study the improvement of software evolvability. This improvement is often referred to by terms such as *restructuring*, *refactoring*, or *re-engineering*. In some cases, even rewriting is used to improve evolvability. If we look at the definitions of software restructuring (Arnold 1989) and software refactoring (Fowler 2000), we can see that they both essentially mean *modification to the internal software structure to make the software easier to understand and modify*. Re-engineering (Chikofsky and Cross 1990) on the other hand means the *examination and alteration of software to reconstitute and implement it in a new form.* Generally re-engineering is used to refer to big system alterations, whereas refactoring and restructuring mean small changes in the code. Often improvement of software evolvability is not studied

[2] Unfortunately, this report has not been accessible to us, as our library was not able to obtain a copy of the report even from abroad. A brief summary of the work can be found from (Pigoski 1996) in page 288.

in isolation. In many re-engineering case studies, improvement in the software evolvability is only one of several goals.

Fourthly, we can study the effect of the current state of software evolvability concerning some other attributes like development efficiency or the number of errors introduced by source code modification. This is perhaps the most widely studied viewpoint to software evolvability. Several well-constructed studies (Bandi et al. 2003; Li and Henry 1993; Rombach 1987) show that using source code metrics for evolvability evaluation can predict the future development effort. Those studies act as motivators for this work.

Table 1 summarises the four viewpoints to software evolvability. The table also provides a general research question that each viewpoint tries to solve. The focus of this paper is on the *evaluation of software evolvability*.

Table 1. Viewpoints to software evolvability

Affecting factors: Which factors can explain the current level of evolvability?
Evaluation: How can we evaluate software evolvability?
Improvement: How can we improve software evolvability?
Effect: What difference does evolvability make (e.g. in terms of development effort)?

2.3 Approaches to Software Evolvability Evaluation and the Scope of the Work

Different approaches to software evolvability evaluation are presented in this section. First, we discuss the level at which the evaluation can be performed. Second, we compare human and machine based evaluation. Third, we look at factors in the software that affect software evolvability. Finally, we present the scope of this work.

2.3.1 Level of evaluation

Software evolvability evaluation can be performed at several levels. We can evaluate evolvability at level of software architecture that consists of systems and sub-systems. We can study the evolvability at sub-system level and look at the package and class structure. Furthermore, we can study the structure of interacting classes that work together to implement certain functionality (e.g. design patterns typically consists of few interacting classes). Still, we can delve even more to the details and study the evolvability of individual classes and methods.

Software evolvability can be studied at various abstraction levels, and this effects the elements we wish to study. For example if we study software at architectural level, we are not well served by looking at the source code, but we should focus on the documentation of the software architecture. Studying the evolvability of an individual method is probably most efficiently done by looking at the actual source code rather than the method documentation. Thus, the level of software evolvability evaluation greatly affects the artefacts we wish to study.

This work focuses on software evolvability at the level of individual source code routines / methods.

2.3.2 Evaluation method

After selecting the level of abstraction where software will be studied, one must also choose the evaluation method. Evaluation can be subjective performed by humans, or objective and performed automatically using program analysis. In this section, we discuss methods for evaluating software evolvability. We identify two fundamentally different evaluation approaches: subjective evaluation performed by humans, and objective metric-based evaluation performed dominantly by program analysis tools. Differences between human evaluation and program analysis are highlighted to conclude the section.

The IEEE standard for software maintenance (IEEE 1998) includes a general process framework for performing software maintenance. In the framework, quality — including software evolvability — is evaluated subjectively by humans in process control points that consist of review, inspection, and verification tasks, and objectively by measures that consist, e.g. of code size and complexity, and error rates. Thus, the standard recognizes the value of subjective human-based evaluation and objective metric-based evaluation, but still leaves most practical issues open.

Widely studied source code or design metrics (Briand et al. 1997; Briand et al. 1999; Chidamber and Kemerer 1994; Halstead 1977; Harrison et al. 1998; Henderson-Sellers 1996; Hitz and Montazeri 1996; Lorenz and Kidd 1994; McCabe 1976; Succi et al. 2005), which can be gathered using program analysis, have traditionally played a big role in evaluating software evolvability. Code metrics have been used and created to form a set of metrics that are able to measure evolvability (Bansiya and David 2002; Chidamber et al. 1998; Szulewski and Budlong 1996). Code metrics have been combined to create polynomial equations whose outcomes give a single measure of evolvability (Coleman et al. 1994; Muthanna et al. 2000). Finally, some researchers have reported success in using code metrics to predict maintenance effort (Grady 1994; Li and Henry 1993). Naturally, the quantitative data should always be interpreted by humans — a fact that undoubtedly introduces some level of subjectivity — but regardless of this there are significant differences between using subjective, qualitative evaluations and objective, quantitative metrics as the basis for evolvability analysis.

Considerably less work has been done studying the use of subjective human evaluation of software evolvability. This topic is interesting because ultimately the developer decides whether software evolvability should be improved or not. Human evaluations are always subjective and thus dependent on the individual doing the assessment. Consequently, one can expect there to be different, even conflicting opinions between the evaluators. This *evaluator effect* can be reduced, e.g. by using evaluation criteria. Subjective human evaluations of software evolvability using code smells can be compared to the judges' evaluations in figure skating or ski jumping competitions.

The differences between program analysis and human evaluations are shown in Figure 1. The figure indicates that program analysis offers a quantitative and objective way to analyse software quality. Human evaluations on the other hand are always more or less subjective, but they offer qualitative information about the software that cannot easily be obtained by program analysis tools. Humans can also consider aspects that are not included in the predefined metrics calculated by tools. Management might be in favour of using program analysis tools, since human opinions could be unreliable. Developers, on the other hand,

can think that metrics are spurious and that you cannot assess the context of each software element by using measurement.

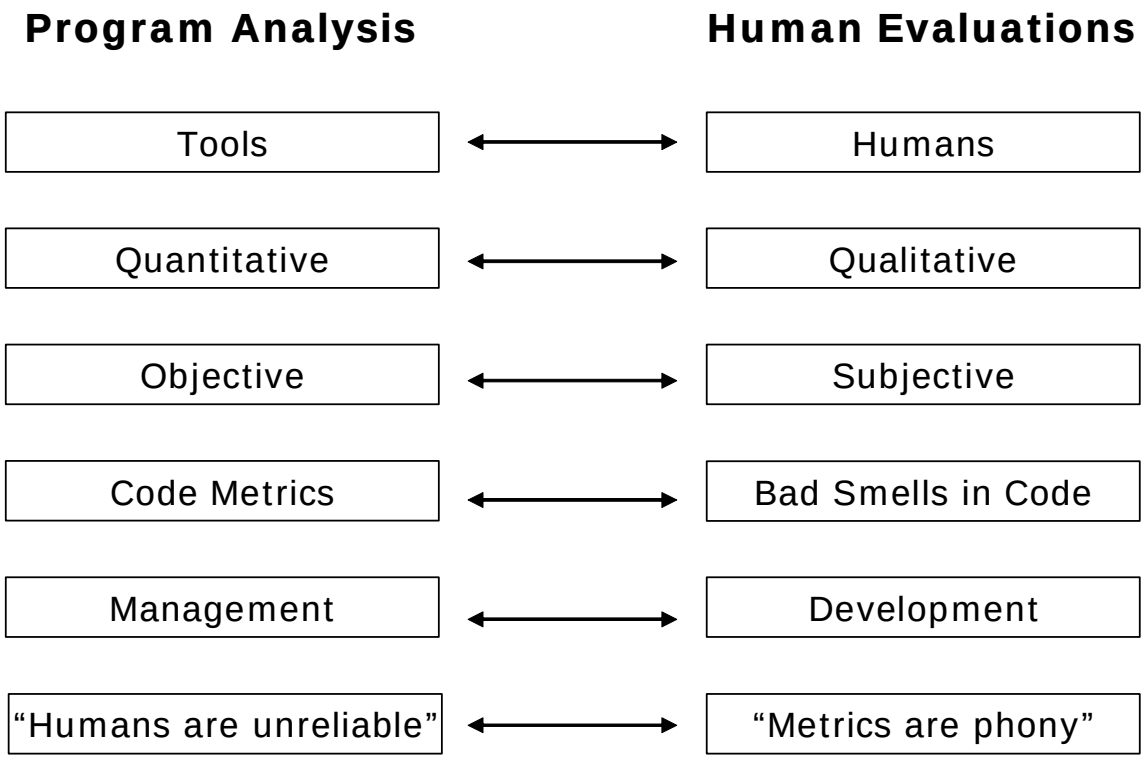


Figure 1. Differences between automatic program analysis and human evaluations

In this study, we focus one the human evaluations because less work has been done in that area. However, we will also contrast the human evaluations with source code metrics to study the link between subjective human evaluations and automatic program analysis based on code metrics.

2.3.3 Bases for evaluation

Software evolvability is also affected by other factors than software structure. Better documentation increases software evolvability. For example, documentation of software architecture raises software evolvability, because it makes software easier to understand and further develop. Even when we study software at the source code level we cannot think that the evolvability could be just determined based on the software structure. Factors like code commenting, naming of code elements, and layout of source code also affect software evolvability.

This work will try to focus studying the software structure at a method level. Our aim is to study software evolvability based on source code structure. However, issues concerning layout, naming, and commenting will be studied because it is not practical or even feasible to study source code structure without concerning these dimension as well.

2.3.4 Scope

Now that we have seen concepts around software evolvability, we can look at the scope of this work:

Evolvability evaluation, including both the evaluation of bad elements in the code and the decision whether to improve the code by refactoring, of software methods based principally on the software structure using human evaluations.

2.4 Human Based Software Evolvability Evaluation

As previously mentioned, several studies have established the link between software evolvability and source code metrics. Recent studies (Balazinska et al. 2000; Ducasse et al. 1999; Kataoka et al. 2001; Simon et al. 2001; Tourwé and Mens 2003) have focused on automatically detecting poor structures in software or have used historical data to detect spots where refactorings had been performed (Maruyama and Shima 1999). For more information on this type of work we point the reader to (Mens and Tourwe 2004). Many of the studies mentioned are actually more focused on evolvability improvement than its evaluation. We have been able to find only a limited number of studies in which subjective evolvability evaluations have been studied or compared with automatic program analysis. In this section, we introduce the relevant prior work we have identified discussing perceived evolvability evaluations.

2.4.1 Subjective evolvability criteria

To make it easier for a software developer to decide whether a certain piece of software needs refactoring (software evolvability improvement) or not, Fowler and Beck (Fowler and Beck 2000) propose a list of 22 bad code smells. Fowler and Beck introduce code smells as a more concrete indication of the need for refactoring than "some vague idea of programming aesthetics". They also claim that no set of precise metrics can be given to identify the need for refactoring. Thus, the code smells can be seen as a compromise between precise source code metrics and totally unguided subjective evaluation. In their experience, Fowler and Beck say that when it comes to making refactoring decisions, no set of metrics rivals informed human intuition. The code smells have been developed based on Fowler's and Beck's industrial experience in several software projects that according to them varied from successful to nearly catastrophic.

Some code smells represent two extremes of the same attribute. For example, the size of a class could be an attribute. Too much of it leads to a smell called "Large Class" and too little to the "Lazy Class" smell. The code smells are some-what vaguely defined. For example, the following is said about the Large Class smell: "When a class is trying to do too much it often shows up as too many instance variables", "...common prefixes or suffixes for some subset of the variables in a class suggest the opportunity for a component.", "As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos and death", "If you have five ten hundred line methods with lots of code in common, you may be able to turn them into five ten line methods with another ten two-line methods extracted from the original", "If your large class is a GUI class, you may need to move data and behaviour to a separate domain object. This may require keeping some duplicate data in both places and keeping the data in sync."

Structures similar to code smells are described by Brown et al. (1998), who discuss software anti-patterns. These anti-patterns describe code problems on class to architectural levels.

Some of them are similar to code smells, e.g., God Class is equal to a Large Class smell and Lava Flow is a synonym for Dead Code. However, the scope of their work is quite wide as they also discuss problems in software processes, badly behaving developers, and many other areas.

The widely recognized software development book "Code Complete" (McConnell 2004) discusses the characteristics of high-quality routines and reasons for creating a class. To summarize, we list the properties of high-quality routines, reasons for creating a class, and classes to avoid as described by McConnell.

- High-quality routines: sufficient reason for existence, contains no code that would benefit from extraction into routines of their own, descriptive names following the naming conventions, high cohesion, low coupling, length that is determined naturally, proper number and usage of parameters.
- Reasons for creating a class: model real-world objects, model abstract objects, reduce complexity, isolate complexity, hide implementation details, limit effects of changes, hide global data, streamline parameter passing, make central points of control, facilitate reusable code, plan for family of programs.
- Avoid classes that: are too big (God Classes), only contain data but no behaviour, only contain behaviour but no data.

A comparison indicates that most of these ideas by McConnell can also be found in the work of Fowler and Beck (Fowler and Beck 2000) that introduced the idea of code smells.

The Air Force Operation Test and Evaluation Center (AFOTEC) pamphlet (AFOTEC 1996) provides a rich set of instructions for evaluating software maintainability. According to the pamphlet, the evaluation is performed by five evaluators that should have no relationship to the software to ensure they are unbiased. As it is seldom humanly possible to evaluate an entire software system, the evaluation is performed on selected source code samples that are representative of the system. The evaluation is performed by agreeing or disagreeing, using a six-point ordinal-scale, with statements that cover different aspects of software maintainability, based on the source code and available documentation. Before the actual evaluation, a calibration run is done to ensure that the evaluators have a "uniform interpretation of how each statement applies to the system". However, the pamphlet particularly stresses that the evaluators should never be forced to change a score they have given. Thus, the purpose is to achieve agreement through discussion on the interpretation of the statements, while the answers are still allowed to vary between evaluators. After the calibration, the team proceeds to the actual evaluation. The statements are grouped into four categories: software documentation, module source listing, computer software unit, and implementation. Some example statements include: Program initialisation is adequately described, Identifier names are descriptive of their use, and Dataflow in this unit is logically organized. To summarize, the AFOTEC pamphlet offers perhaps one of the most complete guides to performing human-based software maintainability evaluations.

All the references discussed in this section provide examples of criteria that could be used when making subjective evolvability evaluations on software. In addition, the AFOTEC pamphlet offers a set of instructions and a process for performing software evolvability evaluation with a team of evaluators.

2.4.2 Studies of subjective evolvability evaluation

Shneiderman et al. (1980 pp. 134-138)) report results from using peer reviews in software code quality evaluation. They conducted three peer review sessions that each had five professional programmers with a similar background and experience as the participants. Each programmer provided one of their best programs which were then evaluated by the four other participants. The review was performed by answering 13 questions on a seven-point ordinal scale. The questions varied from blank line usage and the chosen algorithm to the ease of further development of the program. The results showed that in half of the evaluations three out of four programmers agreed on the subjective evaluations (answers differed by one at the most). Still, in 43,1% of the evaluations the difference between all four evaluators was two or less. The researchers tried to explain this by speculating that the subjects misunderstood the questions or the scale. However, the research does not account for factors such as differences in the developers' opinions about the program design, structure, and style that might explain the results.

Kafura and Reddy (1987) studied the relationship between software complexity metrics and software maintainability. Maintainability was measured using system expert evaluations. However, no details are given on how these evaluations were collected from the individuals and no data is provided of the evaluations. Therefore, it is difficult to assess the study any further. Nevertheless, the researchers conclude that the expert evaluations on maintainability were in conformity with the source code metrics.

Shepperd (1990) validated the usefulness of information flow metrics on software maintainability by collecting the opinions of the maintainers for 89 modules of airspace software that totalled around 30 000 lines of code. Each maintainer of the maintenance team was individually asked to classify each module from one to four on an ordinal scale on the perceived difficulty of some hypothetical maintenance task. In 73% of the individual classifications, the differences per module were one or less and thus the researchers concluded that there was a strong correspondence between the individual ratings. However, as no detailed data is given it is difficult to assess the study in more detail.

Software Engineering Test Lab at the University of Idaho report on the construction of a maintainability index (Coleman et al. 1994; Coleman et al. 1995; Oman and Hagemeister 1994; Welker et al. 1997). In their work, the researchers used source code metrics to create polynomial regression models that measured software maintainability. They calibrated the maintainability models based on how well they correlated with the subjective evaluations of the software maintainers. To do this the researchers acquired source code and the maintainers' opinions on eight industrial software systems ranging from 1000 to 10 000 lines of code (Oman and Hagemeister 1994). After calibration, they performed a validation study where they again acquired opinions and the source code on six industrial systems ranging from 1000 to 8000 lines of code. In the validation study, they also saw discrepancies where one engineer was more lenient and the other ones more critical towards the systems they were evaluating. Although the study (Oman and Hagemeister 1994) does not directly indicate this, it seems that there was only the opinion of a single individual per software system that was used in the initial creation of the metric and the validation performed. Thus, this makes it difficult to study the differences in human maintainability evaluations. After performing tests on several industrial systems, the researchers concluded that the automatic assessment corresponds well to the subjective view of the experts (Welker et al. 1997).

Siy and Votta (2001) studied data of 130 code inspection sessions. Traditionally code inspections have focused on the error data i.e. the problems in the code that will cause failure in operation. Based on their data, Siy and Votta point out that only 18% of the issues identified were true defects, 22% were false positives, and 60% were "soft maintenance-issues". We are interested in the "soft-maintenance issues" since they are quite close to our topic, software evolvability. The researchers grouped together the maintenance issues, which had similar goals. They came up with four groups, namely: Documentation, Style, Portability, and Safety. Documentation contained issues dealing with the documentation in the code. Style issues were related to author's personal programming style, and the suggestions were optional changes to make the code more readable. Portability contained issues that might cause problems in other environments. Safety contained additional checks against things that "should really never happen". The distribution of the maintenance issues were 47% documentation, 46% style, 5% portability, and 2% safety issues. Style issues had a sub-category called clean-up which contained modifications to the code without "changing the meaning of the program". The purpose of those modifications was to prepare the code for subsequent evolution. This sub-category is closest to our scope i.e. subjective software evolvability evaluation based software structure. The study does not have evolvability evaluations, i.e. in scale evolvable not evolvable. Thus, there can be no analysis of interrater agreement, or factors explaining the evaluations. However, it presents important qualitative data of the issues affecting software evolvability.

Kataoka et al. (2002) studied the usefulness of improving software quality with refactoring and report on a comparison between human evaluation and software metrics. According to the researchers, the subjective evaluation of an expert on the effectiveness of refactorings correlated quite well with improvement in the coupling metrics. The drawback in the study is that the data set consists of only five refactoring cases and that only one developer evaluated the effectiveness of the refactorings.

Genero et al (2002) studied the maintainability of UML-class diagrams. The researchers show that subjective evaluation of understandability, analyzability, and modifiability of UML-diagrams correlated (Spearman correlation between 0,529 and 0,943) with various class level metrics. The study had 17 subjects and 28 UML-diagrams. Number of classes and number of attributes had the highest correlations (Spearman correlation between 0,892 and 0,943) with the evaluated quality attributes understandability, analyzability, and modifiability. In a follow up study Genero et al. (2004)[3] show that subjective complexity evaluation has a weak correlation (Kendall's Tau) with time required to understand (0,242), or modify (0,147 and not significant) the UML-diagram. Somewhat larger correlation (0,539) is found between subjective evaluations and objective code metrics based diagram classification. Genero's studies appear to be constructed with empirical rigor, but their reporting is far from perfect. However, these studies are made with UML-diagrams and not with the actual source code. Other shortcoming from our point of view is that they do not provide any results on the interrater agreement of the subjective evaluations.

Mäntylä et al. (2003; 2004) studied code smell evaluations of twelve industrial developers. They discovered the developers make conflicting evaluations of the source code. They also discovered inconsistencies when comparing the evaluations to source code metrics. The weaknesses of the study are that the developers' evaluations were based on recollection of

[3] Unfortunately, this report is missing the section (section 5.2) that contains the results concerning the correlations between subjective and objective complexity. In the report, only a table that contains the correlations is available. Therefore, there is a possibility that we have interpreted their results incorrectly.

the modules they had primarily worked with, and the large size of the evaluated modules (between 15 and 65 thousand lines of code).

2.5 Gap in Existing Work

In this chapter, we have laid the foundation for the work that will be presented in the upcoming chapters. Section 2.1 presents the reasons for using the term software evolvability over the traditional term software maintainability. In Section 2.2 we introduced four viewpoints: factors effecting, evaluation of, improvement of, and the effect of software evolvability. Section 2.3.1 discusses the different levels (architectural level, class level, method level, etc) that can be used when evaluating software evolvability. Section 2.3.2 showed different evolvability evaluation methods focusing on the differences of human evaluations and the source code metrics provided by program analysis. Section 2.3.3 discusses that evolvability is not completely grounded on software structure, but there are other factors, like documentation and layout, effecting on evolvability. Based on our review Section 2.3.4 sets the scope for this work on the software evolvability evaluation grounded mainly on software structure performed by humans at the method level. Then Section 2.4 presents the relevant prior work based on the scope of the work.

Section 2.4 shows that there exists limited amount of studies where subjective evaluation of software evolvability would have been studied in controlled settings, analyzed with statistically solid methods, and with relevant amount of individuals. In addition, many of the prior studies have been made with programming languages that are currently fading away (C, Pascal, and COBOL) from the main stream of software development.

Thus, to better utilize the subjective evolvability evaluations in software development we need to understand subjective evolvability evaluations and its strength and weaknesses. This need is increased by the fact that there are claims and suggestions made on the subjective evaluation of software evolvability without proper scientific studies.

3 Methodology

This chapter presents the research problem of the work that is then further divided to different research questions in Section 3.1. Then two experiments are presented in Section 3.2 that provide the data to answer the research questions. Section 3.3 shows the data analysis methods utilized.

3.1 Research Problem & Questions

Based on the scope of this work, presented in Section 2.3.4, and the limited amount of prior work in this area, presented in 2.4, we selected the following research problem:

Do human evaluations of software evolvability differ from one another, and what are the explaining factors behind the evolvability evaluations?

The research problem clearly has two parts. First, it inquiries whether there are differences between human evolvability evaluations. This problem is called interrater agreement, and it is studied in research question 1 that is presented in Section 3.1.1. Second, it is desirable to explain the factors behind the evolvability evaluations. The second part of the research problem is more interesting as it may explain some of the results of the first part. Therefore, the factors explaining the evolvability evaluations were studied with two different complimenting approaches. Regression analysis was used to provide numeric estimates of the evolvability evaluation variation explained by the different measurable characteristics of the informants or the source code. Qualitative analysis was used to study the rationales of the informants' evolvability evaluations. Researcher questions for regression analysis and qualitative analysis are presented in Sections 3.1.2 and 3.1.3 respectively

3.1.1 Interrater agreement

When studying interrater agreement we must first consider whether there should be interrater agreement between informants. If we ask people for the best way to spend their holiday, we might not expect high interrater agreement, as people are likely to favour different holiday plans. However, we might expect high interrater agreement when we look at the judges of figure skating or ski jumping contests. In those cases, it is essential that these professional judges have some sort of interrater agreement. Although, historically this has not always been the case as the level of agreement between the ski jumping judges has been found non-significant in the past (Vasama and Vartia 1979). Thus, the purpose of our interrater agreement analysis is to study the amount of agreement between different informants on their evolvability evaluations.

Research question 1: *Is there an interrater agreement in subjective evolvability evaluation?*

The *hypothesis* is that for all the three code smells (Long Method, Long Parameter List, Feature Envy) evaluated and for the refactoring decision there should be high interrater agreement.

3.1.2 Factors explaining evaluations — regression analysis

The purpose of regression analysis is to discover how much of the variation in evolvability evaluation, which was the dependent variable, could be explained by the independent variables. There should be two primary sources causing variation in the evaluation. First, the independent variables measuring the characteristics of a software element could affect the evaluations. Second, the independent variables describing the informant who made the evaluations could affect the evaluations.

Research question 2: *How much of the evolvability evaluation of a software element can be explained by the software element and the informant?*

The *hypothesis* is that characteristics of the evaluated element and the evaluators' demographics can explain most of the variation in the evaluations. The software element characteristics are expected to have major impact while the demographic data would have minor impact.

3.1.3 Factors explaining evaluations — qualitative analysis

Qualitative analysis tries to find the factors affecting the evolvability evaluations by looking at the rationales provided by the informants for each evaluation. It differs from the prior two analysis methods by not being a statistical method. In general, qualitative analysis often tries to discover a new hypothesis or a theory that could be tested in future studies, or used to explain the reasons behind the measures.

In this study, qualitative analysis tries to explain the refactoring decisions, and this way it complements the regression analysis. Using regression analysis is based on the assumption that the evolvability evaluations have a linear relationship with the measurable properties of the source code or of the informant. Qualitative analysis goes beyond this and provides a viewpoint inside of an informant's head by studying the rationales for the evolvability evaluation i.e. we can see what each informant thought was good or bad in each software element. Qualitative analysis could also provide some insight to the statistical results of the interrater agreement analysis.

Research question 3a: *What factors act as the rationales for the refactoring decision, could these factors be automatically detected, and what would be the effect of the improvement suggestion factors to common source code measures?*

This research question is of explorative nature and it tries to bring out new qualitative information. Thus, we do not have any hypothesis for the result.

To demonstrate the usefulness of the factors we needed to study the effect these factors to the refactoring decision e.g. if factor X is discovered will this indicate that the refactoring decision will be positive.

Research question 3b: *Can the factors of the rationales predict the refactoring decision, and, if yes what are the most important factors?*

Our hypothesis is that the factors discovered in the rationales should be able to explain most (at least 70%) of the variations in the refactoring decision. Naturally, we cannot hypothesis on the possible factors, as they are the answers to research question 3a.

3.2 Description of the Experiments

3.2.1 Introduction

The goal of the experiments was to study subjective evaluations of software evolvability. Therefore, a repeatable experimental setting was created where the subjective evaluations could be studied in controlled manner.

Two separate experiments with considerable differences were made. We shall refer to them as Experiment A and Experiment B. Both experiments had the same goal as they tried to provide answers to the research problem. Experiment A provided data for the research questions 1 and 2 and Experiment B provided data for research questions 1, 2, and 3.

Experiment A was made first and Experiment B was performed a year later. Experiment B can be though as a successor experiment that tried to provide answers to some of the questions that were created by the results of Experiment A.

Both experiments had the same software elements for the evolvability evaluation. Identical documentation of the software was also provided. However, other things as the experiment material, viewpoints to software evolvability, informants, training, and organization of the experiment differed. The details of these differences are highlighted in the upcoming subsections describing the experiments.

3.2.2 Software under study

To study the subjective evaluations of software evolvability, a small application consisting of 9 classes and approximately 1000 NLOC of code was created. The application was programmed in Java programming language. The application was created solely for the purposes of the experiments. Thus, while creating the application, we purposefully programmed some pieces of the software more poorly than we could have. This way we tried to ensure fluctuation in the software evolvability.

Although, the application was created for this experiment it also has a meaningful task it tries to accomplish. The application is able model a family tree contain relationships between spouses, and parents and children. The family tree modelling was chosen because the domain knowledge required in understanding the application is simple. In fact, all humans can be thought to posses the domain knowledge needed to understand the concepts of close family relationships like father and son.

Screenshots of the application are in Figure 2 and in Figure 3. In the figures, we can see the two views that the application offers. The view shown on Figure 2 can be used to add, delete, and modify people and their information. Figure 3 shows the view where one can modify the relationships between the people.

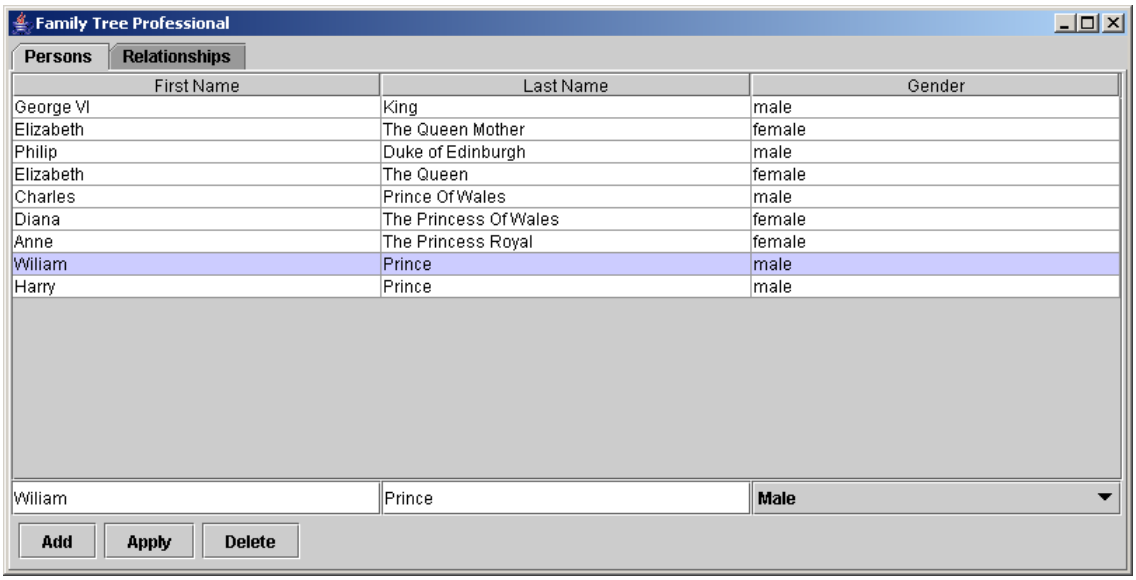


Figure 2. Persons view of the application

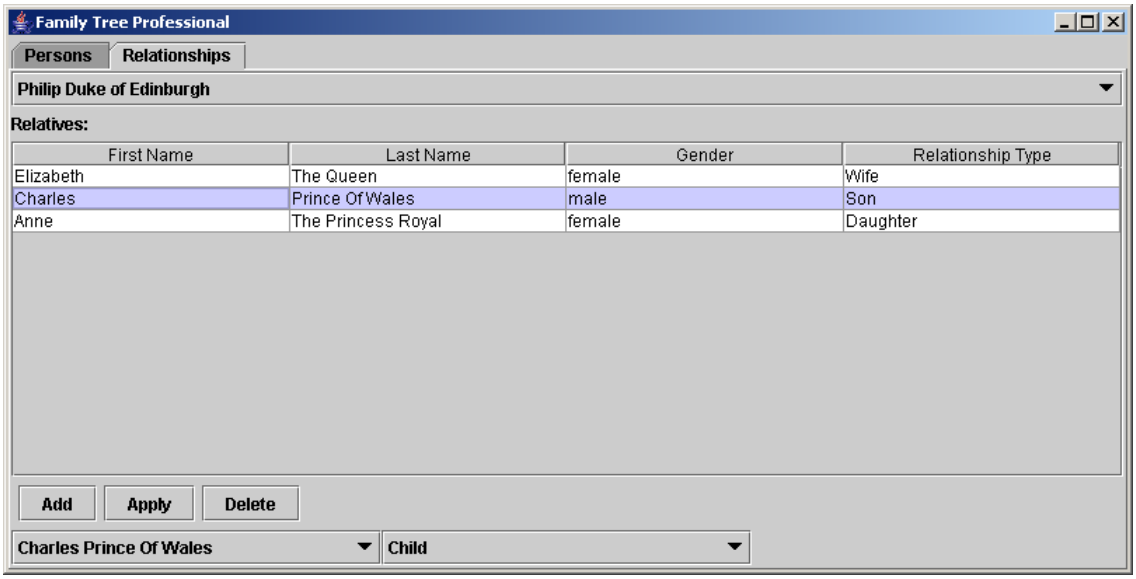


Figure 3. Relationships view of the application

The UML diagram of the application is in Figure 4. From the figure, we can see the classes and their relationships and positions in the package structure. The 10 methods that are visible in the diagram were evaluated in the experiments. FamilyFrame class in the view package is the main GUI-class for the application. PersonTableModel and Relation-TableModel contain data that is showed by FamilyFrame class with java.swing.JTable classes. The data of the application is in the model package classes Person, Relation, RelatioSpouse, and RelationParentChild. The io package contains DiskManager class that is responsible for providing the input output operations for the application. However, currently only reading and writing to disk are supported.

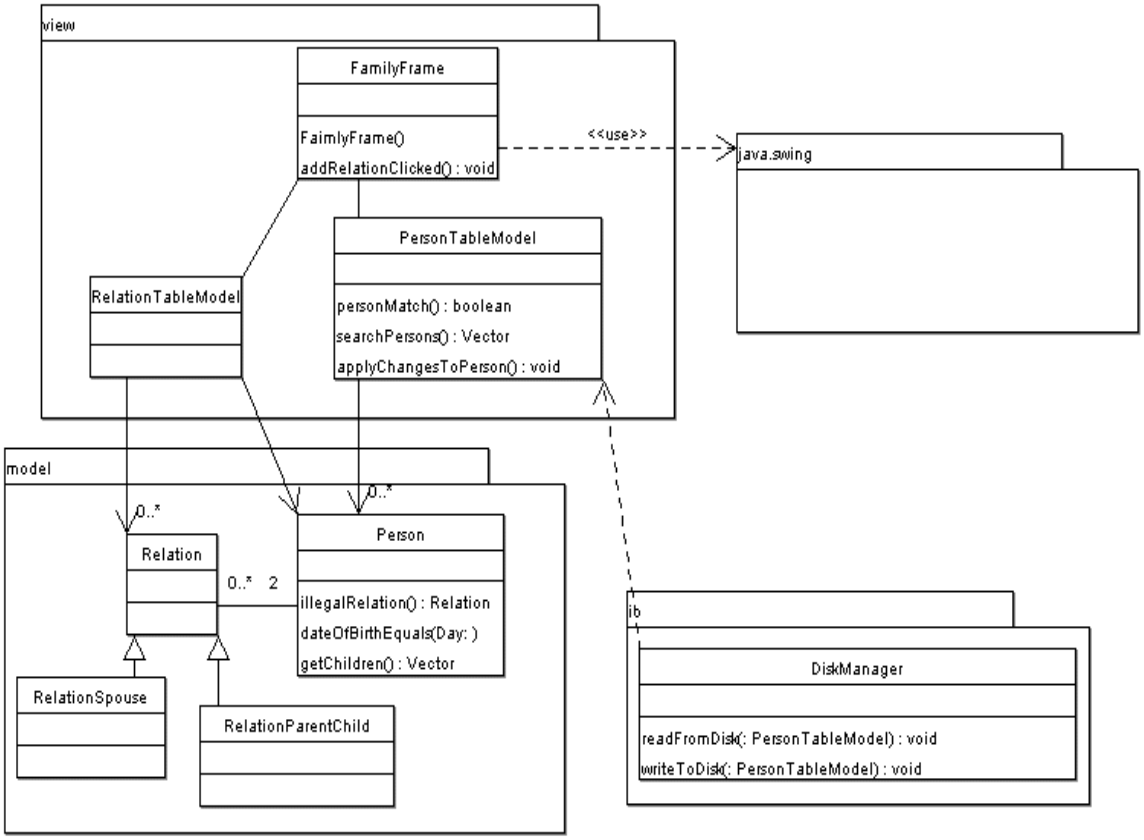


Figure 4. UML-diagram of the application

3.2.3 Viewpoints of subjective evaluations

In both experiments, the subjective evaluations were targeted to the method level. The ten methods that were studied in the experiments from the application are show in Figure 4 and their source code is listed in Listing 1 to Listing 10 in pages 36-49.

Experiment A. To get different viewpoints of the subjective evaluations, four questions of each method were presented. Three of the four questions focused on the existence of the following code smells (Fowler and Beck 2000): Long Method, Long Parameter List, and Feature Envy, which were chosen because they can be studied at the method level. The fourth question asked if the method was in such a bad shape that it should be refactored to remove the smells.

Long method means that a method is too long and tries to perform many possibly unrelated operations. This means that the method has low cohesion and makes it difficult to reuse. Long parameter list means that a method is taking too many parameters. Long parameter lists are difficult to understand and they are continuously changing, as the data needed by the method is varying. Feature envy means that a method is more interested in other classes than the class it is currently located. A method with the Feature Envy smell should be moved to a class that the method is mainly operating with.

Experiment B. In the second experiment no predetermined viewpoints for subjective evolvability evaluations were given i.e. the existence of the smells was not asked. Instead, the informants were asked only if the method was in such a bad shape that it should be

refactored. In addition, the informants were asked to provide their rationale for the previous answer i.e. why the method should be refactored or why not.

3.2.4 The informants of the experiment

The informants of the experiments A and B were students participating in Software Testing and Quality Assurance course at Helsinki University of Technology (HUT) in fall semesters 2003 and 2004. Thus, each experiment had a unique set of similar informants. Below the informants of both experiments are introduced and Table 2 in page 19 summarizes the most important demographic variables. From Table 2 we can see that informants in both experiments were indeed very similar.

Experiment A. In fall 2003, the course had 82 students, with 51 of them that participated in the experiment. We rewarded the participating students by giving them extra points that made up 6 percentage of their total grade. From these 51 students we removed five outliers, whose answers indicated that they had not paid attention, when the instructions were given[4]. After removing these five outliers 46 students were left, and they made up the informants of this experiment.

The informants were studying for a master's degree that requires the minimum of 180 credits. On the average the students had, 115 credits, which means that they had completed approximately two thirds of their studies. The standard deviation of the students' credits was 31,2 with minimum of 34 and maximum of 187,5. Majority of the students (41 of the 46) had between 70 and 158 credits. Years studied by the informants can be seen as irrelevant and even misinforming, because the study times at HUT fluctuate greatly; some students may complete their master's in less than two years starting from undergraduate level while others have studied over ten years. Normally it takes little longer than 6 years to get a master's degree starting from undergraduate or freshman level.

Students at HUT also work during the studies. Therefore, we asked how much work experience the informants had in software development. The mean work experience was 1,5 years with the standard deviation of 1,76. The minimum software development work experience was zero while the maximum was seven years. There were 13 informants (28%) with no software development work experience, but 27 (58%) of the students had year or more work experience in software development. We must make note here that the work experience length between the informants might not be completely accurate, as we do not know whether the informant had been working full time or part time while gaining the work experience in software development.

Majority of the informants (37) were studying in the department of Computer Science and Engineering. Other student were studying in the department of Department of Electrical and Communications engineering (4), department of Automation and System Technology (2), department of Industrial Engineering and Management (2), and department of Engineering Physics and Mathematics (1).

Experiment B. In fall 2004, 37 students participated in the experiment. The students were again rewarded with few extra points except that in Experiment B the extra points were given based on their rationales they provided in the experiment. One student's answers

[4] For example some students claimed that a method had somewhat long parameter list smell, when it had zero parameters. This happened regardless from our threats of forfeiting the extra points if the student would give such incomprehensible and false answers.

were removed as outlier as he/she could not provide reasonable rationales for his/her answers.

The informants were studying for master's degree that requires the minimum of 180 credits. The students had an average of 125 credits. Standard deviation was 38,3 with minimum and maximum of 6 and 199. 86% (31/36) of the students had between 80 and 160 credits. The mean programming related work experience was 1,9 years with the standard deviation 2,91. The median of work experience was one year. The minimum work experience was 0 years and maximum was 15 years. Eight students had no work experience (22%). However, 21 students (58%) had year or more programming related work experience.

No information of the students' department was asked in Experiment B as it was discovered irrelevant in Experiment A. Instead, in Experiment B we asked students grades in HUT's programming courses. This information was not asked in Experiment A. Four students (11,1%) had not taken any programming courses that were asked in the survey. However, 28 (77,8%) students had taken advanced course in object-oriented programming, which is thought after the Basics programming courses.

Table 2. Informants of the experiments

	Experiment A	**Experiment B**
N	46	36
Credits (mean)	115,24	124,54
Work Experience yeas (mean / median)	1,585 / 1	1,875 / 1
Proportion of informant with year or more work experience	58,7%	58,3%

3.2.5 Source code metrics

Source code metrics was used to analyze the evaluated software elements. Here we briefly introduce the metrics and the results of the measurements. The most utilized source code metric is undoubtedly Lines of Code (LOC), and this metric was also used in this study. We adopted the basic version of the Lines of Code measure and calculated it simply as the number of lines in the method including blank and comment lines. Number of Parameters (Par) was measured as it was seen a potential predictor to Long Parameter List smell. Conditional statements are essential in all computer programming, and therefore we decided to measure Cyclomatic Complexity (CC), which is based on interpreting programs as graphs. CC measures the number of linearly independent graphs (McCabe 1976). Cyclomatic complexity is equal to graph's edges minus nodes plus 2. For illustrative example, see Figure 5.

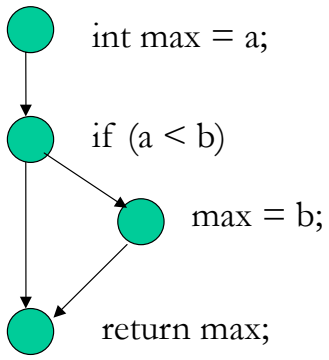


Figure 5. Program with cyclomatic complexity 2 = 4 – 4 + 2

We measured coupling with the Number of Remote Methods called (NORM) and the number of couplings to different objects (CBO). Coupling is created with a field reference or by a method call. This measure was first introduced by Chidamber and Kemerer (Chidamber and Kemerer 1994). Finally, we measured fan-out (FO) that in object-oriented context means the number of reference types used in field declaration, formal parameters, throws declaration, and local variables. No simple types (in this case basic java data types, but also String class) or super types of the object are counted. The results of measuring the methods selected for evaluation are in Table 3.

With this metrics selection, we tried to get a few well-known metrics in order to study their relationships to evolvability evaluations. Researchers have come up with hundreds of other code metrics as well, but have mostly failed to show their relationships with the real world. Thus, the purpose was not to research a vast amount of different metrics, but rather pick few of the most recognized and see if they could be useful in explaining the evolvability evaluations.

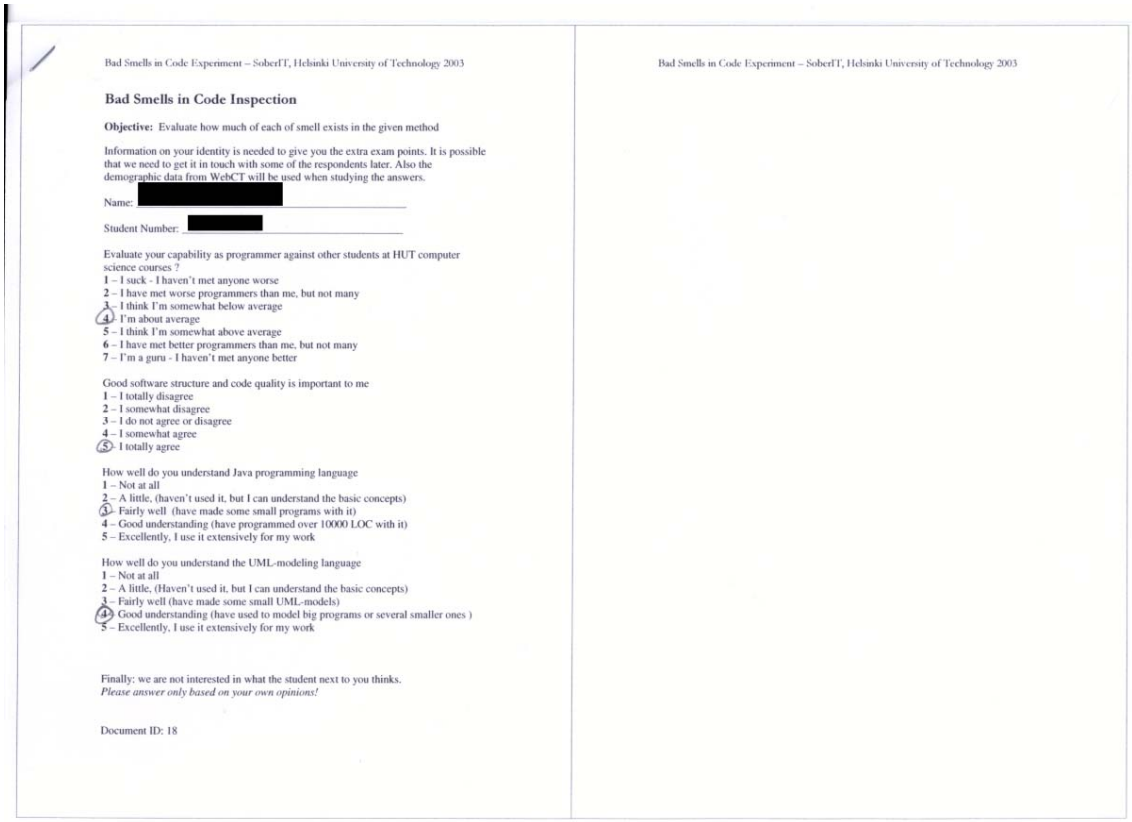
Table 3. Measures from the methods selected for evaluation

Methods	**Metrics**					
	LOC[a]	**Par**[b]	**CC**[c]	**CBO**[d]	**NORM**[e]	**FO**[f]
FamilyFrame.FamilyFrame	84	0	1	13	17	3
Person.dateOfBirthEquals	24	1	2	1	3	1
PersonTableModel.personMatch	19	6	1	1	6	2
DiskManager.readFromDisk	67	1	11	5	16	4
FamilyFrame.addRelationClicked	20	3	6	4	6	1
PersonTableModel.applyChangesToPerson	11	4	1	2	5	1
Person.getChildren	9	0	3	4	7	3
DiskManager.writeToDisk	48	1	7	6	23	7
Person.illegalRelation	46	1	12	5	8	3
PersonTableModel.searchPersons	21	5	3	2	5	4

[a] Lines of code, [b] Parameter, [c] Cyclomatic complexity, [d] Coupling between objects, [e] Number of remote methods called, [f] Fan-out

3.2.6 Experiment material

Experiment A Part of the demographic data was collected at beginning of the course with web-based survey. Questions concerning the rest of the demographics, the source code to be evaluated, and the subjective evaluations about the source code were collected with a survey form. The survey form had 12 pieces of printed A4 paper with the landscape layout so that two pages resided at the each page side by side. To get a better idea of the survey form we refer to Figure 6 and Figure 7, which represent to example pages. However, all the experiment forms were unique as the methods evaluated appeared in random order for each of the informants.



Bad Smells in Code Experiment – SoberIT, Helsinki University of Technology 2003

Bad Smells in Code Inspection

Objective: Evaluate how much of each of smell exists in the given method

Information on your identity is needed to give you the extra exam points. It is possible that we need to get it in touch with some of the respondents later. Also the demographic data from WebCT will be used when studying the answers.

Name: [redacted]

Student Number: [redacted]

Evaluate your capability as programmer against other students at HUT computer science courses ?
1 – I suck - I haven't met anyone worse
2 – I have met worse programmers than me, but not many
3 – I think I'm somewhat below average
4 – I'm about average
5 – I think I'm somewhat above average
6 – I have met better programmers than me, but not many
7 – I'm a guru - I haven't met anyone better

Good software structure and code quality is important to me
1 – I totally disagree
2 – I somewhat disagree
3 – I do not agree or disagree
4 – I somewhat agree
5 – I totally agree

How well do you understand Java programming language
1 – Not at all
2 – A little, (haven't used it, but I can understand the basic concepts)
3 – Fairly well (have made some small programs with it)
4 – Good understanding (have programmed over 10000 LOC with it)
5 – Excellently, I use it extensively for my work

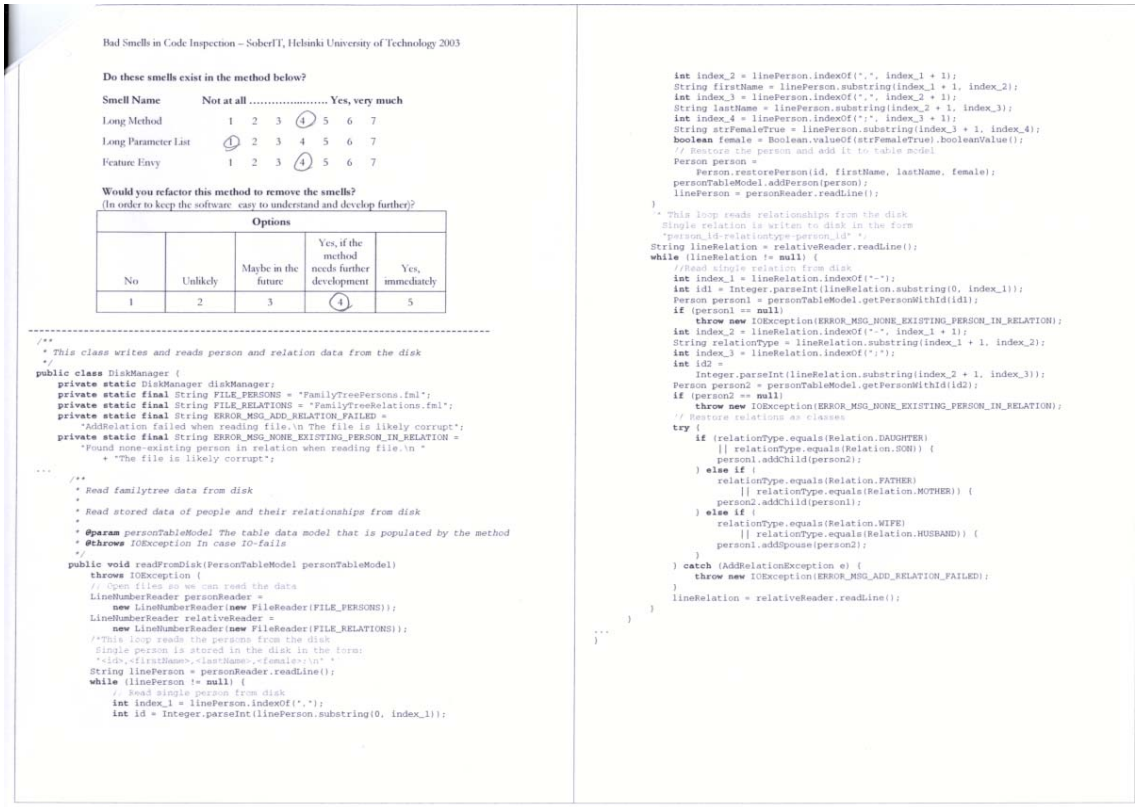
How well do you understand the UML-modeling language
1 – Not at all
2 – A little, (Haven't used it, but I can understand the basic concepts)
3 – Fairly well (have made some small UML-models)
4 – Good understanding (have used to model big programs or several smaller ones)
5 – Excellently, I use it extensively for my work

Finally: we are not interested in what the student next to you thinks.
Please answer only based on your own opinions!

Document ID: 18

Bad Smells in Code Experiment – SoberIT, Helsinki University of Technology 2003

Figure 6. First page of the survey form



Bad Smells in Code Inspection – SoberIT, Helsinki University of Technology 2003

Do these smells exist in the method below?

Smell Name	Not at all ..... Yes, very much
Long Method	1 2 3 (4) 5 6 7
Long Parameter List	(1) 2 3 4 5 6 7
Feature Envy	1 2 3 (4) 5 6 7

Would you refactor this method to remove the smells?
(In order to keep the software easy to understand and develop further)?

Options				
No	Unlikely	Maybe in the future	Yes, if the method needs further development	Yes, immediately
1	2	3	(4)	5

```
/**
 * This class writes and reads person and relation data from the disk
 */
public class DiskManager {
    private static DiskManager diskManager;
    private static final String FILE_PERSONS = "FamilyTreePersons.fml";
    private static final String FILE_RELATIONS = "FamilyTreeRelations.fml";
    private static final String ERROR_MSG_ADD_RELATION_FAILED =
        "AddRelation failed when reading file.\n The file is likely corrupt";
    private static final String ERROR_MSG_NONE_EXISTING_PERSON_IN_RELATION =
        "Found none-existing person in relation when reading file.\n "
            + "The file is likely corrupt";
...
    /**
     * Read familytree data from disk
     *
     * Read stored data of people and their relationships from disk
     *
     * @param personTableModel The table data model that is populated by the method
     * @throws IOException In case IO-fails
     */
    public void readFromDisk(PersonTableModel personTableModel)
        throws IOException {
        // Open files so we can read the data
        LineNumberReader personReader =
            new LineNumberReader(new FileReader(FILE_PERSONS));
        LineNumberReader relativeReader =
            new LineNumberReader(new FileReader(FILE_RELATIONS));
        /*This loop reads the persons from the disk
         Single person is stored in the disk in the form:
         "<id>,<firstName>,<lastName>,<female>;\n" */
        String linePerson = personReader.readLine();
        while (linePerson != null) {
            // Read single person from disk
            int index_1 = linePerson.indexOf(",");
            int id = Integer.parseInt(linePerson.substring(0, index_1));
            int index_2 = linePerson.indexOf(",", index_1 + 1);
            String firstName = linePerson.substring(index_1 + 1, index_2);
            int index_3 = linePerson.indexOf(",", index_2 + 1);
            String lastName = linePerson.substring(index_2 + 1, index_3);
            int index_4 = linePerson.indexOf(";", index_3 + 1);
            String strFemaleTrue = linePerson.substring(index_3 + 1, index_4);
            boolean female = Boolean.valueOf(strFemaleTrue).booleanValue();
            // Restore the person and add it to table model
            Person person =
                Person.restorePerson(id, firstName, lastName, female);
            personTableModel.addPerson(person);
            linePerson = personReader.readLine();
        }
        /* This loop reads relationships from the disk
         Single relation is writen to disk in the form
         "person_id-relationtype-person_id" */
        String lineRelation = relativeReader.readLine();
        while (lineRelation != null) {
            //Read single relation from disk
            int index_1 = lineRelation.indexOf("-");
            int id1 = Integer.parseInt(lineRelation.substring(0, index_1));
            Person person1 = personTableModel.getPersonWithId(id1);
            if (person1 == null)
                throw new IOException(ERROR_MSG_NONE_EXISTING_PERSON_IN_RELATION);
            int index_2 = lineRelation.indexOf("-", index_1 + 1);
            String relationType = lineRelation.substring(index_1 + 1, index_2);
            int index_3 = lineRelation.indexOf(";");
            int id2 =
                Integer.parseInt(lineRelation.substring(index_2 + 1, index_3));
            Person person2 = personTableModel.getPersonWithId(id2);
            if (person2 == null)
                throw new IOException(ERROR_MSG_NONE_EXISTING_PERSON_IN_RELATION);
            // Restore relations as classes
            try {
                if (relationType.equals(Relation.DAUGHTER)
                    || relationType.equals(Relation.SON)) {
                    person1.addChild(person2);
                } else if (
                    relationType.equals(Relation.FATHER)
                        || relationType.equals(Relation.MOTHER)) {
                    person2.addChild(person1);
                } else if (
                    relationType.equals(Relation.WIFE)
                        || relationType.equals(Relation.HUSBAND)) {
                    person1.addSpouse(person2);
                }
            } catch (AddRelationException e) {
                throw new IOException(ERROR_MSG_ADD_RELATION_FAILED);
            }
            lineRelation = relativeReader.readLine();
        }
    }
...
}
```

Figure 7. A sample page with code and the subjective evaluations of the survey form

In addition to the survey forms, the informants also received descriptions of the smells and the UML-figure of the software that can be seen on Figure 4. The description of the smells that were handed out to the informants are in below:

Long Method

- ✓ *Long methods are difficult to understand*
- ✓ *Long methods have low cohesion. Low cohesion means that method tries to do several things rather than performing just a single task.*
- ✓ *Long methods are difficult to reuse*

Long Parameter Lists

- ✓ *Method that has too many parameters suffers from this smell*
- ✓ *Long Parameter Lists are hard to understand*
- ✓ *Long Parameter Lists are constantly changing as more data is needed*

Feature Envy

- ✓ *Method with this smell is more interested in other class(es) than the one it is in*
- ✓ *E.g. if method invokes several getter and setter methods of other classes it is envying data*
- ✓ *Violates the key idea of OO: data and logic are in the same place*

For each method the informants evaluated how much of the smells as described in above existed on the methods. The questions was *Do these smells existing in the method below?* and it was evaluated with ordinal scale from one to seven with one standing for *Not at all* and seven standing for *Yes very much*. In addition to that, a question about refactoring was presented *Would you refactor the method to remove the smells (in order to keep the software easy to understand and develop further).* This was answered with five point ordinal scale where each option had a different meaning: *1-No, 2-Unlikely, 3-Maybe in the future, 4-Yes if the method needs*

to be further development, 5-Yes, immediately. To get a better idea how the questions were presented we refer to Figure 7. In Experiment A the instruction were given during the lecture look for next sub-section for more details.

Experiment B. Experiment B was entirely web-based. The instructions of the experiment consisted of the following: explanation of the task, explanation of refactoring, grading of the experiment, estimated effort required, information about how the experiment information may be used, description of the evaluated software (including screen shots see Figure 2 and Figure 3, UML-model diagram see Figure 4, source code[5], runable application). All this except the description of the software which was shown in Section 3.2.2, can be found in Appendix B. First, the informants answered demographic questions. Then the methods to be evaluated were presented with the question *Would you refactor the method <in question> in order to keep the software easy to understand and develop further?* For the answers there were five options *1-No, 2-Unlikely, 3-Maybe 4-Yes, later when the method needs further development, 5-Yes, immediately*. With the questions, there was a link to source code of the method. The rationale for the evaluation was asked with a question: *Explain your choice? If refactoring is needed, explain what and how the method should be refactored. If the method is OK, explain what desirable qualities the method possesses. If you answered Maybe also give your rationale*. Screen shot of an example evaluation page can be seen in Figure 8.

[5] Interested readers may request the source code of the application from the author

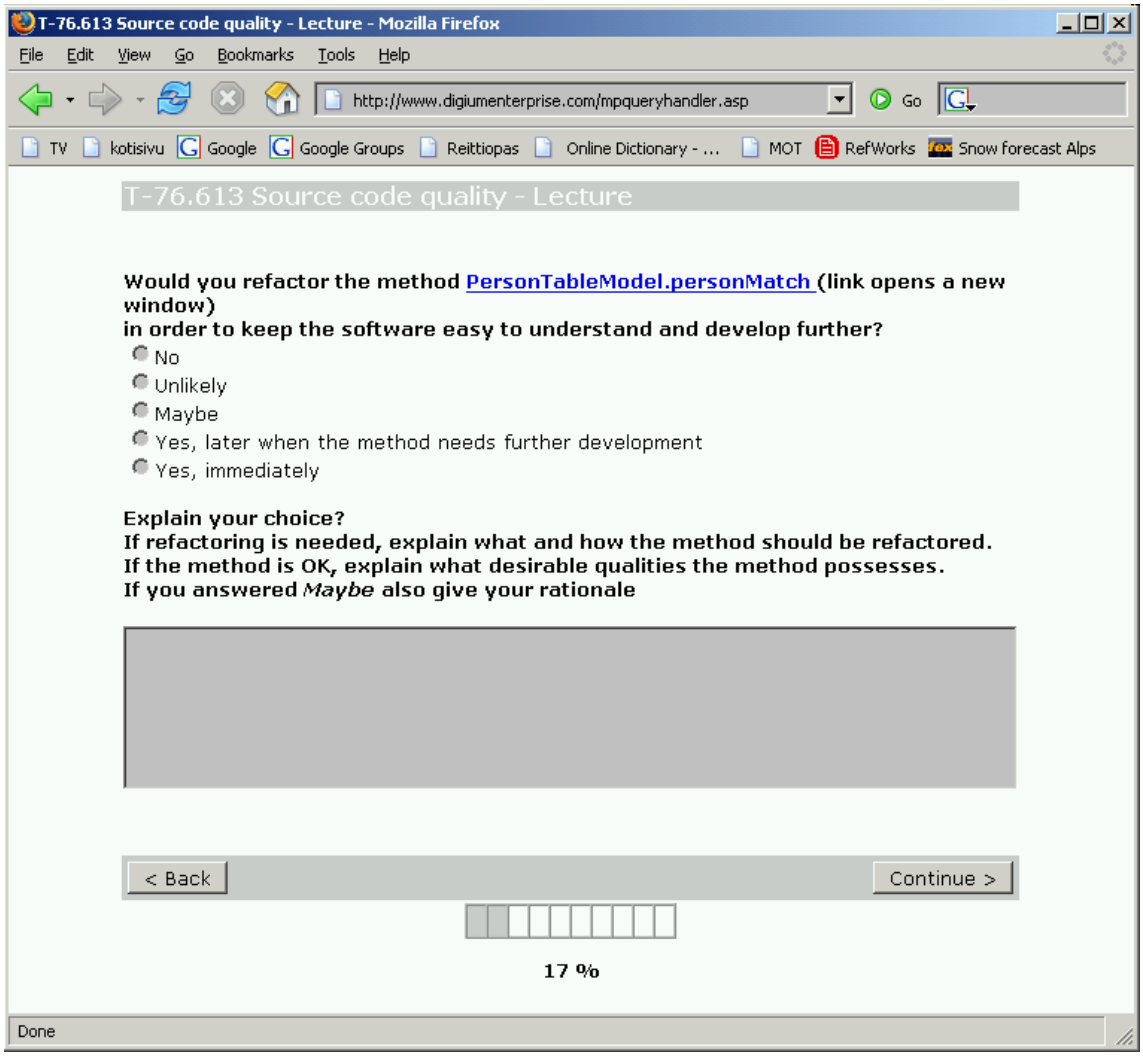


Figure 8. Screen shot of an evaluation page of single method

3.2.7 Training

Experiment A To make sure that all the informants had the required information about the software to be studied, the smells to be evaluated, and the benefits of refactoring, an approximately 20 minute lecture was given on these issues. Additionally, the lecture covered general information about the subjective evaluations as well as some ideas why good software structure could be important, also the exercise organization, and the amount and the criteria for receiving the extra-points was given. The lecture slide outline can be found in Appendix A.

Experiment B. In the second there was no training lecture given before the informants provided their evaluations. As explained in Section 3.2.5 the informants had received information about the software to be studied and the benefits of refactoring. Comparison to Experiment A Experiment B had no smells or any other evolvability flaws that the informants were to search from the software.

3.2.8 During the experiment

Experiment A The experiment was run as a single lecture session, which took altogether about 70 minutes. The session was held in a lecture hall with seats approximately for hundred individuals. First, we gave the informants the training as described in 3.2.7 after that the experiment material as described in 3.2.5 was handed out. Time to evaluate each method was forced to five minutes to guarantee that all the informants used the same amount of effort in evaluating each method. This meant that none of the informants was allowed to proceed until we gave them the instruction to do so. After the experiment, the informants evaluated the usefulness of the experiment (or exercise as it was presented to them). Finally, the informants returned the survey forms.

Experiment B Each informant participated in the experiment through a web-based survey. The informants were able to browse back forward through they answers during the experiment. In the instructions an estimate of the experiment duration was given. During the experiment, the time the informants used to complete the entire survey was tracked by the web-based survey system utilized in the experiment.

3.2.9 Pre-validating the experiment

Experiment A As we would only get a one shot of the experiment with the given group of students, we tried to make the experiment as good as possible. For instance, the experiment material that was introduced in 3.2.5 was reviewed together with our colleagues. Two test runs of the experiment were also arranged with people from our researcher group who had not been exposed to the experiment. Based on the reviews and test runs of the experiments several improvements were made to the experiment material and the organizations. Unfortunately, we do not posses a complete trace for the changes made and the reasons leading to changes. However, here are listed some of the changes. The diversity of the methods was extended by adding a method with none of the smells existing[6]. The reason for this was that at the time the experiment did not have any methods with "perfect" design. In addition, the time to evaluate each method was decreased from six minutes to five minutes since six minutes was found out to be too long in the pre runs of the experiment.

Experiment B This experiment was similar to Experiment A and thus required less pre-validation. After the survey and other material where transformed to html-format and I had tested the experiment, a colleague of mine conducted the experiment. Based on his suggestion few modifications to the experiment were made.

3.3 Data Analysis

This section presents the methods used to analyze the interrater agreement and the factors explaining the evaluations.

3.3.1 Interrater agreement

The Kendall coefficient of concordance (Kendall 1948) which is referred to as *W* or Kendall's *W,* can be used to study the agreement between three or more raters on several

[6] Naturally this is only opinion by me and my colleagues

related samples. Other measures to study interrater agreement in ordinal scale, like Kappa and Kendall's Tau, can measure agreement only between two raters and, therefore, were not applicable. For further information on Kendall's W see ((in press)Legendre 2005; Siegel 1956).

Kendall's *W* tells the amount of interrater agreement by a number between zero and one. If all raters agree *W*=one and *W*=zero implicates the smallest possible amount for the agreement. It is incorrect to use the expression "all disagree" since with *m* observers (m>2) on one-dimensional data it is impossible to completely disagree on anything (Kendall 1948). With m=two it is possible to disagree completely on rankings and this results W=zero and Spearman correlation of minus one (in pressLegendre 2005). Completely random data with m=two would result W=0.5 and Spearman correlation 0. When m, the number raters, increases the expectation value of W decreases such that W is 1/m. Additionally, the statistical significance of the agreement is needed. Small significance means that there is at least partial concordance among the raters. Kendall's *W* significance is equivalent to the value produced by Friedman's analysis of variance test.

Interrater agreement can also be studied in other fields e.g. judges of sports like ski jumping and figure skating should be able to reach high interrater agreement. For a reference the Kendall's *W* was calculated from the first round results of ski-jumping world cup competition[7] held in Oberstdorf, Germany at 29th December 2004. In that occasion the five judges evaluating 50 jumps achieved Kendall's *W* 0,888 and asymptotic significance 0,000.

3.3.2 Factors explaining evaluations – regression analysis

Regression analysis was used to study how the method characteristics and the informants' demographics affected the evaluations. There were various features concerning the method characteristics and the respondents' background, and the data came in various scales (nominal, ordinal, interval), and therefore could not be analyzed using classical linear regression. Thus, the data was analysed using categorical regression that is available for SPSS™ software. Categorical regression is founded on optimal scaling, which turns nominal and ordinal variables into linear variables (Meulman 1998). The categorical data analysis methods for SPSS™ have been developed in Data Theory Group from Leiden University, The Netherlands.

Categorical regression performs better than traditional ordinal regression on data sets that have limited amount of observations, too many variables, and too many values per variable (Meulman and Heiser 2001). This was also observed in this study when ordinal regression in SPSS™ was used to analyze the data. For instance in most cases no reliable statistical analysis could be performed because the answers could not provide all the possible combinations for the 7-point ordinal scale. Optimal scaling provides another advantage against traditional methods, which is the ability to create a combined regression models where different types of independent variables can exist simultaneously and also the dependent variable can belong to any of the three categories nominal, ordinal, and continuous (Meulman and Heiser 2001 pp. 8). Thus, categorical regression based on optimal seemed the most suited option for this type of statistical analysis.

[7] Results are available from http://www.fis-ski.com/

Several regression models were created for each of the predicted (dependent) variables, which were the evaluations of the existence of three code smells and the refactoring decision. The first model (called the MetDem model) used the source code metrics of the methods and the demographics of the informants as predictor (independent) variables. The second model contained only the demographics and the third model only the source code metrics of the method. In Experiment A, an additional model was constructed that used the smell evaluations as predictor variables when predicting the refactoring decision.

3.3.3 Factors explaining evaluations – qualitative data

In Experiment B, the informants were asked to provide rationales for each refactoring decision. This resulted textual answers ranging from four words to seven sentences per evaluated software element per informant. The total number of separate qualitative comments was 360. Qualitative research greatly varies in research strategies and types, look for example Miles and Humberman (1994) pages 6-7. Thus, there is no single analysis method or process that could be taken and used straight of the shelf. Based on the ideas gathered from (Moilanen and Roponen 1994; Miles and Huberman 1994; Seaman 1999) the coding process presented in below and Atlas.ti program was used to analyze the textual data. During the analysis of the textual answers, the demographic data of the informants and the evolvability evaluations were not studied to prevent possible bias they could have caused.

First, all answers were read through and coding paradigm was used to recognize the type of information they presented. In this work, we shall call the recognized types of information as *topics* instead of *codes* to avoid confusion with coded text passages and program code. Literature (Moilanen and Roponen 1994; Miles and Huberman 1994; Seaman 1999) suggests that topics can be either preformed or postformed. Preformed topics mean that before the text is analyzed a list of topics is created. Postformed topics mean that topics are created during the coding process. Often in practice, some of the topics are preformed and some of them are postformed. In this study, mostly postformed topics were used. However, as the author had studied code problems (e.g. bad code smells (Fowler and Beck 2000)) extensively, this meant that some topics were actually preformed in the authors mind although they were not explicitly written down before the coding process was started.

Second, after all the answers had been analyzed and assigned with one or more topic, an analysis of topics was performed. This meant that topics were renamed to better reflect their contents, and some topics were merged as there had been several topics with different names reflecting identical meaning.

Third, all the answered were again read through and recoded with new topics where needed. This was done to have constant topic usage across the answers. Some topics were formed during analyzes of the final answers, and, thus, such topics would not have appeared in the rest of the answers. Short memos of answers for each method that the informants had evaluated were also written in this stage.

Fourth, each topic was individually analyzed by looking at the text passages that each topic pointed. This was done in order to discover patterns inside the topics, to detect wrongly categorized text passages, and to understand better what each topic actually meant. During this process, topics were also merged and split up when ever it was necessary.

Fifth, the relationships of the topics were studied by grouping them in to topic families that gathered similar topics.

Sixth, topics with similar targets but different types (look Section 4.3.1 and Figure 9 for definition of topic type and target) were combined to get the total amount of persons who were pointing out similar problems. For example if an informant was pointing out that method is too long and was also saying that it should be split up, this comment had both a complain and improvement suggestion. However, the target of the comment, method size, remains the same.

3.3.4 Factors explaining evaluations – regression analysis with qualitative data

Categorical regression that was introduced in Section 3.3.2 was used to perform regression analysis on the qualitative data. Categorical regression was chosen because the dependent variable was in ordinal scale and the qualitative topics are best interpreted as ordinal scale data. These topics were discovered when performing qualitative data analysis as described in Section 3.3.3. The different topic types can be found in results chapter Section 4.3

- Either a text contained a topic or it did not. Thus ordinal binary scale
- With OK topic there was three different levels plus a case where no OK topic was found. Thus ordinal scale from 0-3
- There were also different topic families e.g. complaints and improvement suggestions concerning software structure. These topic families contained several topics. A value of a topic family was obtained by counting how many of its topics are in the text. One might have been temped to call this an interval scale, but since it was based ordinal values we shall also interpret is as ordinal.

Different regression models were studied. First, a model where the refactoring decision was explained by the OK topics and topic family called negatives was created. It is called *Simple Model*. OK topic was split up three levels 1, 2, and 3, where three is highest meaning that method was praised in some way and one being the lowest indicating an OK, but also a minor complains made concerning the method. Topic family negatives contained all comments whose type was a complain or an improvement suggestion.

Second, a model containing the negative topics grouped on topic families structure, documenting, and visual representation was created. It is called *Target Model*. An extended version of this model was also created by adding OK topics to this model. It is called *Target&OK Model*. In this model topic families structure, documenting, and visual representation have slightly fewer topics than topic family negatives because some topics inside topic family negatives were too general to be categorized into any of the topic families. This analysis aimed to discover how much impact did each topic family (structure, documenting, and visual representation) have on the refactoring decision.

Finally, just pure topics were used to predict the refactoring decision. This model would reveal the individual topics that had the greatest affect on the refactoring decision. This regression model is called *Pure Topics Model*.

4 Results

This chapter has four sections that present the results of the study. First, we look at the Interrater agreement in Section 4.1. Section 4.2 studies the factors explaining evolvability evaluations with regression analysis. Section 4.3 presents results of qualitative factors affecting the refactoring decision. Finally, Section 4.4 uses the qualitative factors as bases for regression analysis in explaining the refactoring decision.

4.1 Interrater Agreement

Table 4 shows the results of the interrater agreement analysis. The informants had a high agreement on evaluations concerning the Long Method and Long Parameter List smells. The agreements concerning the Feature Envy smell the refactoring decision and are considerably weaker. However, all *W* values are significant indicating that the informants had at least some level of agreement. *W* values in the refactoring decision for both experiments are close to each other. The number of rankings available varied from 44 to 46 in Experiment A. In Experiment B, there were 36 rankings. For all the cases, the number of evaluated objects was 10.

Table 4. Interrater agreement

Question	N	Kendall's *W*	p-value
Exp A - Long Method	46	0,777	0,000
Exp A - Long Parameter List	46	0,816	0,000
Exp A - Feature Envy	44	0,238	0,000
Exp A - Refactoring	45	0,353	0,000
Exp B - Refactoring	36	0,397	0,000

4.2 Factors Explaining Evaluations - Regression Analysis

This section studies the sources of variation in the evolvability evaluations and tries to predict the evaluations using regression analysis.

4.2.1 Long Method

Regression models for the Long Method smell are in Table 5. From the table, we can see that the MetDem model, consisting on the source code metrics and informants' demographics, explains 74,6% of the evaluations. The MetDem model details revealed that the source code metrics were the most important predictors. This can be also seen in Table 5 where the Metric model predicts 71,2% of the evaluations. However, the Demographic model, containing the information about the informants' background, is not able to explain the evaluations effectively.

Table 5. Long Method regression models

Model	Adjusted R Square	p-value
MetDem	0,746	0,000
Metric	0,712	0,000
Demographic	0,012	0,231

As it seems that, the metrics rather than the demographics explained most of the Long Method evaluations it made sense to study them in more detail. In Table 6 we can see the predictor variables of the Metric model and their standardized beta's, p-value of the F-values, and the correlation with the predicted variable.

Table 6. Predictors in the metric model for Long Method smell evaluations

Code Metric	Standardized Beta	p-value	Correlation
Number of Parameters	-0,108	0,002	-0,509
Lines of Code	0,738	0,000	0,815
Cyclomatic Complexity	0,114	0,001	0,474
Fan Out	0,008	0,866	0,480
Number of Remote Methods	0,171	0,014	0,700
Coupling Between Objects (C&K)	-0,190	0,001	0,624

From Table 6 we can see that Lines of Code was the most important predictor in the model. The Lines of Code in this case meant that a single line is a single line of code regardless of the space usage or comments in the method. NLOC[8] was also tested in the regression model, but it performed slightly poorer, although the difference was merely marginal. Other metrics had only a subsidiary effect in the Metric model, but most of them had a high correlation with the Long Method evaluations. This indicates that a reasonably good regression model could be created even without the lines of code metric, and when this was tested the Metric model without Lines of Code was able to explain 61,8% of the evaluations. In that model number of remote methods (std beta 0,531), cyclomatic complexity (std beta 0,335), and coupling between objects (std beta 0,255) were the best predictors

4.2.2 Long Parameter List

The regression models for the Long Parameter List smell are in Table 7. The results are similar to the results of the Long Method smells. We can see in Table 7 that the MetDem model explained 77,6% of the evaluations, and the Metric model was almost as good explaining 76,1% of the evaluations.

[8] lines of code with out comments and blank lines so that one statement equals a line of code

Table 7. Long Parameter List regression Models

Model	Adjusted R Square	p-value
MetDem	0,776	0,000
Metric	0,761	0,000
Demographic	0,054	0,003

Further analysis of the Metric model in Table 8 shows that the Number of Parameters is the most important predictor. It has very high correlation with the predicted variable. Number of Remote Methods and Fan-out also have some impact in the regression model. However, it seems likely that the effect was caused more by the limited amount of methods evaluated rather than by real effect. Additionally, only the Number of Parameters metric has a positive correlation with the predicted variable.

Table 8. Predictors in the Metric Model for Long Parameter List smell evaluations

Code Metric	Standardized Beta	p-value	Correlation
Number of Parameters	0,807	0,000	0,857
Lines of Code	0,095	0,074	-0,466
Cyclomatic Complexity	-0,170	0,000	-0,424
Fan Out	0,229	0,000	-0,228
Number of Remote Methods	-0,287	0,000	-0,441
Coupling Between Objects (C&K)	0,061	0,246	-0,494

4.2.3 Feature Envy

Regression models for the Feature Envy smell are in Table 9. The MetDem model was able to explain only 29,8% of the evaluations. The Metric Model explained only 9,8% of the evaluations. Thus, with Feature Envy it appears that the predictors failed in predicting the Feature Envy smell evaluations. Consequently, there is no need to look at the individual models any further.

Table 9. Feature Envy regression models

Model	Adjusted R Square	p-value
MetDem	0,298	0,000
Metric	0,098	0,000
Demographic	0,054	0,003

4.2.4 Refactoring decision

In refactoring decision regression analysis, there were data from both experiments. The regression models of the refactoring decision can be seen in Table 10 and Table 11. In Table 10 we can see that SmeMetDem model which consists of smell evaluations, source code metrics and demographics, explained 66,5% of the evaluations. The most significant contributors in the SmeMetDem model are the smell evaluations. The Metric model explained 31,9% of the evaluations. Finally the Demographic model was not effective at all explaining only 8,7% of the evaluations.

Table 10. Refactoring decision regression models in Experiment A

Model	Adjusted R Square	p-value
Exp A - SmeMetDem	0,665	0,000
Exp A - Smell	0,618	0,000
Exp A - MetDem	0,435	0,000
Exp A - Metric	0,319	0,000
Exp A - Demographic	0,087	0,000

Experiment B did not have a SmeMetDem or Smell model because no questions concerning the code smells were asked in that experiment. As can be seen from Table 11, the MetDem model explained 28,4% of the evaluations. Further analysis showed that most of the explanative power came from source code metrics. The Metric model explained 26,1% of the evaluations alone while the demographic model failed to be effective. The comparison to Experiment A shows that the Demographic model in Experiment A performed slightly better, but this is likely to be caused by the larger set of demographic variables than a real effect.

Table 11. Refactoring decision regression models in Experiment B

Model	Adjusted R Square	p-value
Exp B – MetDem	0,284	0,000
Exp B – Metric	0,261	0,000
Exp B – Demographic	0,036	0,026

Details of the Smell Model from the Experiment A can be seen in Table 12. From the table we can see that all the smell evaluations were important when predicting the refactoring decision. The Long method smell evaluations were the most significant contributors, but even the evaluations of the Feature Envy smell contributed significantly to the Smell model.

Table 12. Predictors of the Refactoring decision in Smell Model in Experiment A

Code Metric	Standardized Beta	p-value	Correlation
Long Method	0,598	0,000	0,514
Long Parameter List	0,469	0,000	0,280
Feature Envy	0,360	0,000	0,518

In Table 13 we can see the predictors of the Metric model predicting the refactoring decision in both experiments. In the table, we can see that the lines of code measure had the highest beta in the regression equation, and it also had the highest correlation among the source code metrics with the refactoring decision. In both experiments, Coupling between objects (CBO) was a suppressor term, which traditionally would indicate a reduction in the likelihood of refactoring. However, this was more likely caused to by the multicollinearity that source code metrics have with each other. This is supported by the facts that CBO had positive correlation with the refactoring decision and that Lines of Code and CBO had high correlation with each other (Person correlation 0,830, significance 0,000). Therefore, we cannot claim that increase in CBO would decrease the refactoring need.

Table 13. Predictors of the Refactoring decision in Metric model in Experiments A and B

	Experiment A			Experiment B		
Predictor	**Std. Beta**	**p**	**Correlation**	**Std. Beta**	**p**	**Correlation**
Number of Parameters	0,397	0,000	0,130	-0,133	0,032	-0,230
Lines of Code	0,805	0,000	0,381	0,923	0,000	0,453
Cyclomatic Complexity	-0,003	0,957	0,157	-0,073	0,230	0,233
Fan Out	0,063	0,417	0,215	0,057	0,534	0,168
Number of Remote Methods	0,011	0,916	0,157	-0,162	0,197	0,277
Coupling Between Objects	-0,299	0,001	0,272	-0,500	0,000	0,246

4.2.5 Summary

From the results, we can see that Lines of Code is the best and most important predictor in the Long Method smell evaluations, regardless of the fact that other problems of the long method were introduced to the informants as described in Section 3.2.5. However, well performing regression model was created even without using the lines of code metric. This is easy to explain when we know that many source code metrics are correlated with each other. In Metric regression model with out Lines of Code metric Number of Remote Method, Cyclomatic Complexity, and Coupling Between Object rose as the most important

predictors. From those three metrics, only Cyclomatic Complexity did not have significant correlation with the lines of code metric.

With Long Parameter List smell evaluations the Number of Parameters was the most significant predictor. There is no reason to believe that any other metric would be good predictors of this problem, because the smell evaluations had significant correlation with other metrics. The evaluations on Feature Envy smell, however, could not be explained by the dependent variables in the regression model.

The smell evaluations in Experiment A were able to provide the best explanation for the refactoring decision explaining 61,8% of the variation. The used source code metrics could not effectively explain the refactoring decision explaining only 31,9% and 26,9% of the variation in experiments A and B respectively.

4.3 Factors Explaining Evaluations – Exploring Qualitative Data

This section will explore the rationales of the informants' evolvability evaluations. This is qualitative research and exploratory by nature.

4.3.1 Topics and topic families

Topics and topic families can tell us what different types of problems the informants recognized in the software elements they were evaluating. The total number of topics was 67. Some topics had only one or two text references when some topics had over 70 references. Before going through each individual topic in detail, we study the topic families of the answers to get an overview of the qualitative data provided. Figure 9 shows the topic families represented with Unified Modelling Language (UML), which is originally designed to model object oriented programming elements.

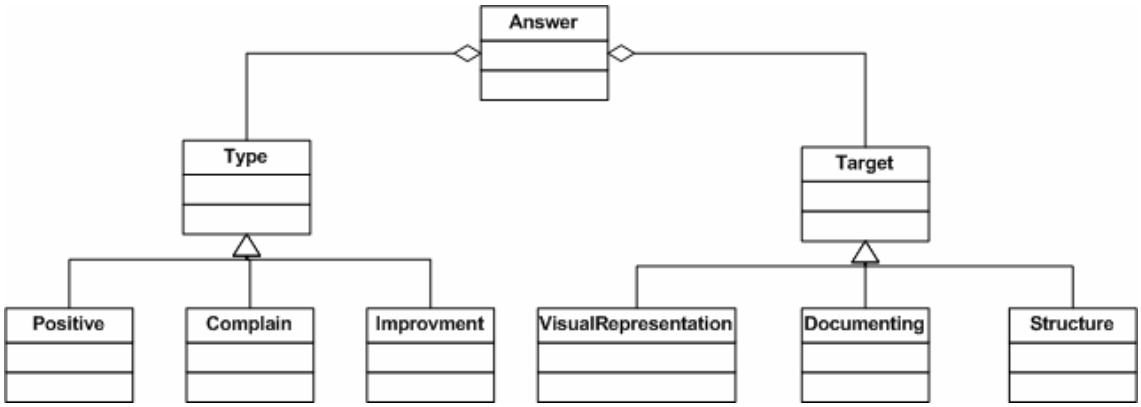


Figure 9. Topic families of the answers

In the figure we can see that Answer could consists of zero or more *Types* and *Targets*. *Types* reflected the type of answer whether it was a positive comment, complain, or an improvement suggestion. *Target* represented the part of the software element the answer was concerned of. Visual representation means issues in representing the software topic that can make it easier for a human to read. Topics on visual representation contained comments for example on blank line usage, indentation, and splitting up method parameters to their own lines. Documenting means communicating information about a software element

to the reader. In source code level[9], such documenting is most often done by the names of the software elements and source code comments that provide clarification when naming cannot alone explain the source code's intent. In this study, topic family documenting consisted of source code naming or comments. Topic family structure had comments on the software structure. Topic family structure included comment whether a certain method should be split up to smaller methods, or should the parameters be passed inside an object or separately.

Figure 9 is purposefully missing the reasons for each complaint. For example if informant was complaining about the structure of a software element one might wish to know the reason for this complaint. In this study, the reasons for the answers were however given in the task description, which was the evolvability of the software (the form of the question can be read from Section 0 Figure 7). However, few answers also commented other issues like performance and correctness, although they were not specifically asked for.

By comparing the scope of the work, which was presented in Section 2.3.4, and the textual answers we can see that some of the informants' answers actually are outside of the original scope of the work since they have specified other *targets* than structure and other reasons than evolvability. However, as this part of the work was qualitative it made sense to study also those answers as it helps us in explain the refactoring decision.

4.3.2 Qualitative data in detail

It is difficult to understand the value of qualitative data by looking at it from the distance. Therefore, we go thorough all the software methods that were evaluated by the informants. We look at interesting topics and patters that can only be discovered by studying individual software elements. We analyze topics that appear more often in each method than in other methods. We also study topics that according to our opinion should have received more mentions from the informants. We consider how certain topics could be automatically detected. We use complain topics as bases for refactorings to remove those topics and reflect the refactorings effects to source code metrics. We list source code and the most used topics for each method. All topic frequencies per software element can be found in Appendix C. The number of topics in Appendix C does not equal the number of informants because a single informant may be responsible for one or more topics. The same holds for topics that are selected for further analysis of each method in tables from Table 14 to Table 23.

The first method evaluated by the informants can be seen in Listing 1. The topics selected for further analysis of the method are in Table 14. From the table we can see that 17 informants were not happy with the length of the return statement. Nine of the informants were not happy with the comments. They indicated that parameter comments are missing or that within code comments are confusing. Seven people claimed that method is OK. Only two informants rightfully pointed out that method is inside a wrong class (method should be in Person class instead of PersonTableModel), and only one informant considered the number of parameters to be too high. It is interesting that only few people pointed out the two latter comments, although those issues can be considered as important as the length of the statement in the method.

[9] In higher levels of representation, visual documenting is used.

Table 14. Selected topic frequencies in method PersonTableModel.personMatch

Topic name	Number of informants	Percentage (out of 36)
Long Statement or Statement Split	17	47%
Comments improve	9	25%
OK	7	19%
Move method	2	6%
Long Parameter List	1	3%

Listing 1. Method PersonTableModel.personMatch

```
/**
 * PersonTableModel.personMatch()
 *
 * Does the person given as the first argument match
 * with the search criteria given as arguments
 */
private boolean personMatch(
        Person person,
        String firstName,
        String lastName,
        int gender,
        Calendar dateOfBirth,
        Calendar dateOfDeath) {
    return
    //Compare firstname
    (firstName == null || person.getFirstName().equals(firstName)) &&
    //Compare lastname
    (lastName == null || person.getLastName().equals(lastName)) &&
    //Compare gender
    (gender == person.getGenderAsInt() || gender == -1) &&
    //Compare day of birth
    (dateOfBirth == null || person.dateOfBirthEquals(dateOfBirth)) &&
    //Compare day of death
    (dateOfDeath == null || person.dateOfDeathEquals(dateOfDeath));
}
```

A measure could have been used to detect the long statement, for example, one could calculate the number of conditionals in a single statement or used other kind of statement weight or length measure. Many informants suggested splitting the statement to several if – else clauses, which would have slightly increased the lines of code in the method. However, lines of code is inaccurate measure, in Listing 1 for example all method parameters are split to their own separate rows and combining them to same line would decrease this measure when only effecting the method's visual representation. Therefore, we suggest measure such as statement count i.e. calculate all statement ending with ";"-mark and special statements containing conditionals expressions like while, for, if. This type of measure would be immune to changes in the method's visual representation. Splitting the long statement in Listing 1 would dramatically increase the statement count, because currently the method has only has one statement and after splitting the statement there could be easily over ten statements. The effect of statement split to would also increase the cyclomatic complexity (McCabe 1976), which measures the lineally independent execution graphs inside the method.

Many informants pointed out that parameters are not commented, which could have been measured or detected. Naturally, the quality of comments could not be measured or detected automatically. The effect of commenting all parameters would increase the number of comment lines, which is traditionally seen as a positive measure[10]. The need for moving the method to another class can be detected with coupling measure measuring the maximum number of remote method calls to one class. Moving method to other location would have decreased coupling.

The second evaluated method can be seen in Listing 2. The topics selected for further analysis of the method are in Table 15. From the table we can see that 17 informants thought that the method is too long or that it should be split to smaller methods. Eight informants said that a certain part of the method's algorithm should be changed. Seven out of these eight were pointing out that the duplication removal is not needed if duplicate relationships are not added to the data structure in the first place, or that the duplicate removal should be better programmed. One remaining informant suggested using Java's object serialization functionality instead of self programmed file format. Seven persons pointed minor details that were classified under Minor Coding Conventions. They include changing the for-loop to a while-loop, using iterators in all loops, or processing or introducing variables in different part of the method. Again, there are seven informants saying that the method is OK.

Detecting the long method could be easily achieved with lines of code or statement count measure. The extract method refactoring would decrease the lines of code or statement count measure. However, it would increase the number of methods per class that is used to measure class size. The effects of performing the change algorithm refactoring to get rid of duplicate removal functionality would shorten the method. Changing the method's algorithm to use Java's serialization routines would dramatically shorten and simplify the method as whole objects and object relationships could be stored with a single command[11]. Automatic detection for the need of the change algorithm refactoring is not feasible.

Table 15. Selected topic frequencies in method DiskManager.writeToDisk

Topic name	Number of informants	Percentage (out of 36)
Long Method or Extract Method	15	42%
Change Algorithm	8	22%
OK	7	19%
Minor Coding Conventions	7	19%

[10] However, recently some people have presented ideas that comments can be seen as symptom of poorly programmed code e.g. using comments to explain the programs behavior rather than programming it be evident.

[11] However, there are issues involved in serialization that make compatibility of different file versions questionable. Therefore it is not necessarily an optimal solution in this case.

Listing 2. Method DiskManager .writeToDisk

```
/**
* Write the family tree data to the disk
*
* Write the people and their relationships to disk
*
* @param personTableModel
*            This data model contais the people we will write to disk
* @throws IOException
*/
public void writeToDisk(PersonTableModel personTableModel)
        throws IOException {
    // Open files for writing
    FileWriter fileWriterPerson = new FileWriter(FILE_PERSONS);
    FileWriter fileWriterRelations = new FileWriter(FILE_RELATIONS);
    // Create temp variables
    Vector persons = personTableModel.getPersons();
    Vector relations = new Vector();
    Vector relationsToBeRemoved = new Vector();
    //Write persons to disk according to format:
    //"<id>,<firstName>,<lastName>,<female>;\n"
    for (Iterator iter = persons.iterator(); iter.hasNext();) {
        Person person = (Person) iter.next();
        fileWriterPerson.write(person.getId() + ",");
        fileWriterPerson.write(person.getFirstName() + ",");
        fileWriterPerson.write(person.getLastName() + ",");
        fileWriterPerson.write(person.isFemale() + ";\n");
        relations.addAll(person.getRelationships());
    }
    //Find duplicate Relationships
    for (int i = 0; relations.size() > i; i++) {
        Relation rel1 = (Relation) relations.elementAt(i);
        for (int j = i + 1; relations.size() > j; j++) {
            Relation rel2 = (Relation) relations.elementAt(j);
            if (rel2.equals(rel1)) {
                relationsToBeRemoved.add(rel2);
            }
        }
    }
    //Remove duplicate relationships
    for (Iterator iter = relationsToBeRemoved.iterator(); iter.hasNext();) {
        relations.remove(iter.next());
    }
    //Write relations to disk to format:
    //"person_id-relationtype-person_id"
    for (Iterator iter = relations.iterator(); iter.hasNext();) {
        Relation relation = (Relation) iter.next();
        fileWriterRelations.write(relation.getPerson1().getId() + "-");
        fileWriterRelations.write(relation.getRelationType(relation
                .getPerson1())
                + "-");
        fileWriterRelations.write(relation.getPerson2().getId() + ";\n");
    }
    //Clean up and close the files and streams
    fileWriterPerson.flush();
    fileWriterPerson.close();
    fileWriterRelations.flush();
    fileWriterRelations.close();
}
```

The third evaluated method can be seen in Listing 3. The topics selected for further analysis of the method are in Table 16. Majority of the informants said that the method is OK. OK-topic actually has three levels 1, 2, and 3. Three is the highest meaning the method was praised in some way, and one is the lowest indicating acceptance of the method but also having a minor complaint. In this method 8 out of 22 OK-topics where level 3, and only one was level one OK-topic. Four people pointed out that gender

handling with string-variable and equals-command is a bit strange. Four people also pointed out that the called method person.setFemale is badly named and it should be changed to person.setGender. In this case, four informants' answers indicated that they did not completely understand what was going on in the method. Two of them where wondering the intention of the last expression which uses variable rowIndex twice as an argument in a single method call[12]. One was suggesting that separate methods should be created for setting the different parameters. This makes no sense since this is user interface code that is executed right after user has pressed apply in the user interface. The last informant, that lacked understanding, commented in her answer that maybe she did not completely understand the structure of method.

Automatic or metrics based detection of any of the problems of this method seem impossible. We cannot see a way to detect automatically the naming problems or the gender handling problems.

Table 16. Selected topic frequencies in method PersonTableModel.applyChangesToPerson

Topic name	Number of informants	Percentage (out of 36)
OK	22	61%
Gender handling	4	11%
Naming other methods	4	11%
Lack understanding	4	11%

Listing 3. Method PersonTableModel.applyChangesToPerson

```
/**
 * Update the selected persons data
 * @param firstName New first name
 * @param lastName New last name
 * @param gender New gender
 * @param rowIndex The row that contains the person to be updated
 */
public void applyChangesToPerson(
        String firstName,
        String lastName,
        String gender,
        int rowIndex) {
    Person person = (Person) persons.elementAt(rowIndex);
    person.setFirstName(firstName);
    person.setLastName(lastName);
    person.setFemale(gender.equalsIgnoreCase("female"));
    fireTableRowsUpdated(rowIndex, rowIndex);
}
```

The fourth method evaluated can be seen in Listing 4. The topics selected for further analysis of the method are in Table 17. This method has many similar characteristics as the method in Listing 2. In both methods, the method length or extracting method to smaller pieces was the most mentioned topic. The second frequent topic in both cases was the

[12] This part is actually partly confusing, but it is correctly coded and the informants could have checked this from Java SDK API

Change Algorithm. In this method 11 out 12 informants with Change Algorithm topic pointed out, that part of the code that is reading the data from disk could be more easily implemented with using: Java's String Tokenizer functionality, Java's regular expression functionality with String.split method calls, Java's Serialization functionality, or using loop with array instead of several index variables. Six informants thought the method was OK. Although, there were no OK level 3 comments, which indicates the lack of praises in the OK comments. Four informants also thought that the number of temporary variables declared and used in the method should be reduced.

The metrics and refactoring analysis for this method is in many ways similar to the method in Listing 2. However, some differences need to be discussed. The effect of change algorithm refactoring to start using Java's regular expression or String Tokenizer would made individual statements simpler and it would reduce the code lines needed. The temporary variable reduction refactoring could be detected by measuring the number of temporary variables inside the method. The effect of this refactoring would reduce the number variables, but it would not decrease the number of code lines or number statements.

Table 17. Selected topic frequencies in method DiskManager.applyChangesToPerson

Topic name	Number of informants	Percentage (out of 36)
Long Method or Extract Method	16	41%
Change Algorithm	12	33%
OK	6	16%
Temps reduce	4	11%

Listing 4. Method DiskManager.readFromDisk

```
/**
 * Read familytree data from disk
 *
 * Read stored data of people and their relationships from disk
 *
 * @param personTableModel
 *            The table data model that is populated by the method
 * @throws IOException
 *             In case IO-fails
 */
public void readFromDisk(PersonTableModel personTableModel)
        throws IOException {
    // Open files so we can read the data
    LineNumberReader personReader = new LineNumberReader(new FileReader(
        FILE_PERSONS));
    LineNumberReader relativeReader = new LineNumberReader(new FileReader(
        FILE_RELATIONS));
    /*
     * This loop reads the persons from the disk Single person is stored in
     * the disk in the form: " <id>, <firstName>, <lastName>, <female>;\n"
     */
    String linePerson = personReader.readLine();
    while (linePerson != null) {
        // Read single person from disk
        int index_1 = linePerson.indexOf(",");
        int id = Integer.parseInt(linePerson.substring(0, index_1));
        int index_2 = linePerson.indexOf(",", index_1 + 1);
        String firstName = linePerson.substring(index_1 + 1, index_2);
```

```
        int index_3 = linePerson.indexOf(",", index_2 + 1);
        String lastName = linePerson.substring(index_2 + 1, index_3);
        int index_4 = linePerson.indexOf(";", index_3 + 1);
        String strFemaleTrue = linePerson.substring(index_3 + 1, index_4);
        boolean female = Boolean.valueOf(strFemaleTrue).booleanValue();
        // Restore the person and add it to table model
        Person person = Person.restorePerson(id, firstName, lastName,
                female);
        personTableModel.addPerson(person);
        linePerson = personReader.readLine();
    }
    /*
     * This loop reads relationships from the disk Single relation is writen
     * to disk in the form "person_id-relationtype-person_id"
     */
    String lineRelation = relativeReader.readLine();
    while (lineRelation != null) {
        //Read single relation from disk
        int index_1 = lineRelation.indexOf("-");
        int id1 = Integer.parseInt(lineRelation.substring(0, index_1));
        Person person1 = personTableModel.getPersonWithId(id1);
        if (person1 == null)
            throw new IOException(
                    ERROR_MSG_NONE_EXISTING_PERSON_IN_RELATION);
        int index_2 = lineRelation.indexOf("-", index_1 + 1);
        String relationType = lineRelation.substring(index_1 + 1, index_2);
        int index_3 = lineRelation.indexOf(";");
        int id2 = Integer.parseInt(lineRelation.substring(index_2 + 1,
                index_3));
        Person person2 = personTableModel.getPersonWithId(id2);
        if (person2 == null)
            throw new IOException(
                    ERROR_MSG_NONE_EXISTING_PERSON_IN_RELATION);
        // Restore relations as classes
        try {
            if (relationType.equals(Relation.DAUGHTER)
                    || relationType.equals(Relation.SON)) {
                person1.addChild(person2);
            } else if (relationType.equals(Relation.FATHER)
                    || relationType.equals(Relation.MOTHER)) {
                person2.addChild(person1);
            } else if (relationType.equals(Relation.WIFE)
                    || relationType.equals(Relation.HUSBAND)) {
                person1.addSpouse(person2);
            }
        } catch (AddRelationException e) {
            throw new IOException(ERROR_MSG_ADD_RELATION_FAILED);
        }
         lineRelation = relativeReader.readLine();
    }
}
```

The fifth method that was evaluated can be seen in Listing 5. The topics selected for further analysis of the method are in Table 18. Most (27/36) of the informants said the method was OK and six of them made a level 3 praising OK comment. Four informants pointed out poor readability. Two of the four mentioned that the children.add method call was not easy to understand. The other two informants did not specify why they thought that the method was not easy to understand. Answers with topics Reorganize internally (need of reorganization of the method internals) were saying variable relation should be declared outside of loop or that the iterator should be initialized before the loop structure. Two people indicating the need for changing the algorithm suggested using different data structure than Vector. One informant pointed out that with large amount of relatives, the search becomes slow and therefore the parent should constantly hold a list of its children. The minor coding convention topic included issues of changing the for-loop to a while-loop and to use curly braces with the if-statement.

The problems of poor readability and the need to change algorithm cannot be detected automatically. However, we can detect and measure the variable declaration inside of a loop. The effects of the proposed refactorings to source code metrics would be very small if not inexistent.

Table 18. Selected topic frequencies in method Person.getChildred

Topic name	Number of informants	Percentage (out of 36)
OK	27	75%
Readability poor	4	11%
Reorganize internally	3	8%
Change algorithm	3	8%
Minor coding conventions	3	8%

Listing 5. Method Person.getChildren

```
/**
 * Get children of this person
 * @return Vector containing the children
 */
public Vector getChildren() {
    Vector children = new Vector();
    for (Iterator iterator = vecRelations.iterator(); iterator.hasNext();) {
        Relation relation = (Relation) iterator.next();
        if (relation.isParent(this))
            children.add(((RelationParentChild) relation).getChild());
    }
    return children;
}
```

The sixth method that was evaluated can be seen in Listing 6. The topics selected for further analysis of the method are in Table 19. Over half of the informants said that the method was too long and should split up i.e. some suggested creating graphical user interface (GUI) components for the two tabs in their own methods. This can be seen a bit surprising since GUI code often contains long methods, as there seldom is possibility to reduce the amount of code by generalization. Half of the informants said that the method was not very easy to read or comprehend. These informants characterized the code as long, messy, awful, and lacking proper grouping, but also pointed out that GUI code often looks like this. The informants complaining about the layout said that there should more empty lines to provide better grouping. Surprisingly, there were two informants saying that the method's internal grouping is good (OK) Four people said the method overall was OK, although two of these four had level 1 OK-topics indicating some complain accompanying the OK statement. However, there was one person praising the method and thus characterized as level 3 OK:

> *Metodi oli selkeä. Person ja Relation paneeleihin koskevat koodit oli selkeästi jaksotettu ryhmiinsä ja myöhemmin jos tarvitaan lisää poimintoja, on ne helposti lisättävissä oikealle paikalleen koodia.*

Here is an English translation of that comment

The method was clear. Code concerning Person and Relation panels was clearly grouped, and if there is later, a need to add functionality it can be easily added to the correct place in the code

Again, in this method the method size could be easily measured with lines of code or statement count measure. However, cyclomatic complexity measure would not have indicated any problems since the method does not have any conditional statements. Performing the extract method refactoring would decrease the size of this method, but it would increase the number of methods in this class. Nothing can be said about detecting or refactoring poor readability, as it is very general statement. The poor layout could have been detected by the lack of blank lines compared to the lines of codes in the method. The effect of improving layout would be seen in code lines versus blank lines ratios.

Table 19. Selected topic frequencies in method FamilyFrame.FamilyFrame

Topic name	Number of informants	Percentage (out of 36)
Long Method or Extract Method	22	61%
Readability Poor	18	50%
Layout poor	12	33%
OK	4	11%

Listing 6. Method. FamilyFrame.FamilyFrame

```
/**
 * Constructor of the main GUI class in the application
 * This method creates GUI components for the application
 */
private FamilyFrame() {
    super("Family Tree Professional");
    setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
    JTabbedPane tabbedPane = new JTabbedPane();
    JPanel personPanel = new JPanel(false);
    JPanel relativePanel = new JPanel(false);
    personPanel.setLayout(new BorderLayout());
    relativePanel.setLayout(new BorderLayout());
    // Two tabs are used. personPanel for showing all the persons
    // relativePanel for showing all the relations
    tabbedPane.addTab("Persons", null, personPanel, "Shows the persons");
    tabbedPane.setSelectedIndex(0);
    tabbedPane.addTab("Relationships", null,
            relativePanel,"Show the relationships");
    //Table for Person PersonTable
    personTableModel = new PersonTableModel();
    personTable = new JTable(personTableModel);
    personTable.setPreferredScrollableViewportSize(new Dimension(500, 70));
    //Table for Relations RelationTable
    relationTableModel = new RelationTableModel();
    relationTable = new JTable(relationTableModel);
    relationTable.setPreferredScrollableViewportSize(
            new Dimension(500, 70));
    //Create the scroll pane and add the PersonTable to it.
    JScrollPane scrollPane = new JScrollPane(personTable);
    personPanel.add(scrollPane, BorderLayout.CENTER);
    //Create the scroll pane and add the RelativeTable to it.
    JScrollPane scrollPane2 = new JScrollPane(relationTable);
    relativePanel.add(scrollPane2, BorderLayout.CENTER);
    //Create the panel where person data can be edited
    //Add it personPanel
    JPanel personPanelEdit = new JPanel();
    personPanelEdit.setLayout(new BorderLayout());
```

```
    //Text fields for editing person
    JPanel personFieldPanel = new JPanel();
    personFieldPanel.setLayout(new GridLayout(0, 3));
    personFieldPanel.add(jtfFirstName);
    personFieldPanel.add(jtfLastName);
    personFieldPanel.add(jcbGender);
    personPanelEdit.add(personFieldPanel, BorderLayout.NORTH);
    //Buttons for editing person
    JPanel buttonPanelPerson = new JPanel(new FlowLayout(FlowLayout.LEFT));
    buttonPanelPerson.add(jbAddPerson);
    buttonPanelPerson.add(jbApplyPerson);
    buttonPanelPerson.add(jbDeletePerson);
    personPanelEdit.add(buttonPanelPerson, BorderLayout.SOUTH);
    personPanel.add(personPanelEdit, BorderLayout.SOUTH);
    //Create the panel where relationships can be edited.
    //Add it to relativePanel
    JPanel relativePanelEdit = new JPanel();
    relativePanelEdit.setLayout(new BorderLayout(0, 1));
    JPanel relativeFieldPanel = new JPanel();
    relativeFieldPanel.setLayout(new GridLayout(0, 3));

    relativePanel.add(createTopRelativePanel(), BorderLayout.NORTH);
    relativePanelEdit.add(relativeFieldPanel, BorderLayout.NORTH);
    //Buttons for editing relationships
    JPanel buttonPanelRelative =
        new JPanel(new FlowLayout(FlowLayout.LEFT));
    buttonPanelRelative.add(jbAddRelation);
    buttonPanelRelative.add(jbApplyRelation);
    buttonPanelRelative.add(jbDeleteRelation);
    buttonPanelRelative.add(jcbRelationship);
    relativePanelEdit.add(buttonPanelRelative, BorderLayout.CENTER);
    //RelativeInfoPanel, where the person's info
    //who is edited/added/deleted as relative is added
    JPanel relativeInfoPanel = new JPanel();
    relativeInfoPanel.setLayout(new GridLayout(0, 3));
    jcbFullNameRelativeToBeAdded =
        new JComboBox(
                new DefaultComboBoxModel(personTableModel.getPersons()));
    relativeInfoPanel.add(jcbFullNameRelativeToBeAdded);
    relativeInfoPanel.add(jcbRelationship);
    relativePanelEdit.add(relativeInfoPanel, BorderLayout.SOUTH);
    relativePanel.add(relativePanelEdit, BorderLayout.SOUTH);
    //Add the tabbed pane to this window.
    getContentPane().add(tabbedPane, BorderLayout.CENTER);
    addListeners();
    readDataFromDisk();
}
```

The sixth method that was evaluated can be seen in Listing 7. The topics selected for further analysis of the method are in Table 20. Over half of the informants were happy with the method. Some informants pointed out that it is good thing the method is using helper method inside the if-statement. However, there only two informants indicated that the method should have been using parameter object. The informants were correct as parameter object would make this method immune for the changes in the criteria (currently passed as parameter list) used in searching and matching a person. Six informants that told the method would need more comments were mostly pointing to the fact that parameters had not been commented. Two informants indicated they would change the method arguments to be in a single line. Finally, six informants pointed out that variable mathingPersons is spelled incorrectly.

The need for method parameter comments could have been automatically detected. The effect of this change would increase the lines of comment measure. Spelling problems with the single variable could not be detected; the use of spelling checker in this case is not feasible because source code contains variables names like firstName, which would be

flagged by any regular spellchecker. Only special source code spell checker (if existing) could point out this problem. The problem concerning the layout of method arguments could be detected and even automatically enforced. This change in layout would reduce the lines of code in the method but it would not affect the statement count measure. It is easy to measure the number of the parameters and state that parameter object could be used. However, this measure is not detecting the fact that all the parameters are used as a group when they are referenced inside a method call. Referring them only as a group and not making separate references to them makes the parameter object even more tempting because there would be no need to make any method calls or field references to the parameter object. Thus, the coupling to the parameter object would be minimal. To detect the need for this refactoring we would have to somehow measure that parameters are only used as a group and they are not referenced individually. Performing this with program analysis is possible, but not necessarily very simple.

Table 20. Selected topic frequencies in method PersonTableModel.searchPersons

Topic name	Number of informants	Percentage (out of 36)
OK	21	58%
Comments Add	6	17%
Spelling & grammar	6	17%
Layout method arguments	2	6%
Parameter Object	2	6%

Listing 7. Method PersonTableModel.searchPersons

```
/**
 * Get the persons that match the arguments
 * @return Vector contain the matching persons
 */
protected Vector searchPersons(
        String firstName,
        String lastName,
        int gender,
        Calendar dateOfBirth,
        Calendar dateOfDeath) {
    Vector mathingPersons = new Vector();
    // Loop throught the persons and check if they match
    for (Iterator iter = persons.iterator(); iter.hasNext();) {
        Person person = (Person) iter.next();
        if (personMatch(person,
                firstName,
                lastName,
                gender,
                dateOfBirth,
                dateOfDeath)) {
            mathingPersons.add(person);
        }
    }
    return mathingPersons;
}
```

The eighth method that was evaluated can be seen in Listing 8. The topics selected for further analysis of the method are in Table 21. Twelve informants were saying that the final

return statement is too long and it should be split up. This is similar to the complains concerning the long statement problem in Listing 1 that also had a long return statement. Seven informants said that the method is OK. Seven informants also indicated that readability of the method is poor. Two of these seven did not indicate the problem in more detail, but complained about the horrible appearance and said the method was not easy to understand. The remaining five out of the seven were pointing that the return statement is unreadable. Three out of the five layout complainers pointed out that the return-statement is not correctly intended. The remaining two layout complainers complained about the lack of blank line usage. Four informants would have extracted a new method from this method. Three of them would make own methods from the comparisons of day, month, and year. One pointed out that the bug avoidance should have been separated in to its own routine, which is good idea, but the return value of that method should be an array or other data structure. Three informants spotted that this method is almost identical to the method just below this method called Person.dataOfDeathEquals and suggested combining them.

Table 21. Selected topic frequencies in method Person.dataOfBirthEquals

Topic name	Number of informants	Percentage (out of 36)
Long Statement or Statement Split	12	33%
OK	7	19%
Readability Poor	7	19%
Layout Poor	5	14%
Extract Method	4	11%
Duplication	3	8%

The measurement and the effect of refactoring a long statement were already discussed in Listing 1. In addition, we have already discussed how the poor layout can be automatically detected and corrected. The detection of extract method refactoring in this method would have been difficult. One informant indicated that the bug fix should be extracted to own method. Detecting this would have been challenging, and the only idea that comes in to mind is to try to search for in-method comments that contain words "bug" and "fix" near to each other. Three informants answered that they would extract own methods out of the checks of day, month, and year. These three informants clearly indicate that they prefer using small utility methods for condition testing. Automatically supporting this kind of coding style would require a tool that could distinct between the utility methods and the worker methods. Then the tool would need to detect statements in the worker methods that do not utilize a utility method but still makes use of identical conditional comparisons. Creating such detection tool would not be straightforward operation. The effects of extract method refactorings to source code metrics have been previously discussed. Duplicate or clone source code detection is an established research area look for example (Balazinska et al. 2000; Ducasse et al. 1999) for more references. It seems likely that the duplication found by the informants could have been automatically detected. The effect of removing the duplication would reduce the class size measured as lines of code or number of statements,

but it would increase the method count in the class if both methods would retain their presence and a new method would be created to contain the shared functionality.

Listing 8. Method Person.dataofBirthEquals

```
/**
 * Does the given birth day match with this persons birthday
 *
 * @param moment The moment that is checked as birthday.
 * It is possible to specify only only month, year or day in the moment variable
 * This way we can check whether his person was born in July for instance
 *
 * @return Whether the given moment includes the birthday
 */
public boolean dateOfBirthEquals(Calendar moment) {
    if (moment == null)
        return false;
    //Ugly hack to fix feature/bug by SUN, which causes
    //all calender fields to get set, when Calender.get(Field) is called
    Calendar day1 = (Calendar) moment.clone();
    Calendar day2 = (Calendar) moment.clone();
    Calendar day3 = (Calendar) moment.clone();
    Calendar bday1 = getDateOfBirth();
    Calendar bday2 = getDateOfBirth();
    Calendar bday3 = getDateOfBirth();
    return (
            //Check day of month
            (!day1.isSet(Calendar.DAY_OF_MONTH)
                    || (bday1.get(Calendar.DAY_OF_MONTH) == day1.get(Calendar.DAY_OF_MONTH)))
                    &&
                    //Check month
                    (!day2.isSet(Calendar.MONTH)
                            || (bday2.get(Calendar.MONTH) == day2.get(Calendar.MONTH)))
                            &&
                            //Check year
                            (!day3.isSet(Calendar.YEAR)
                                    || (bday3.get(Calendar.YEAR) == day3.get(Calendar.YEAR))));
}
```

The ninth method that was evaluated can be seen in Listing 9. The topics selected for further analysis of the method are in Table 22. Two thirds of the informants said that the method was OK. Seven informants pointed out that comments should be improved, two of these were saying that the usage of Person parameters should made clearer. The rest of the informants who complained about comments pointed that the Javadoc should tell what the method does. Six people said that the naming of Person parameters should be made more evident. However, none of them proposed how it should be done, and one of them even admitted that he could not come up with names that are more descriptive. This indicates that descriptive names are important, but not in all cases easy to come up.

Table 22. Selected topic frequencies in method FamilyFrame.addRelationClicked

Topic name	Number of informants	Percentage (out of 36)
OK	24	67%
Comments improve	7	19%
Naming improvement variables	6	17%

Detecting the problems listed for this method in Table 22 would be very difficult. Detecting the problem in the comments is impossible because it requires interpretation of the comments. Poor variable naming could be perhaps detected by searching variables having a number as last part of their name. This could be used as an indication of lack of self-documenting names. However, it seems likely that such solution would provide a great amount of false positive results.

Listing 9. Method FamilyFrame.addRelationClicked

```
/**
 * This method is executed, when user adds a relation between to persons.
 * E.g. addRelation button is clicked
 * @param person1 Person participating in the relation
 * @param person2 Other person participating in the relation
 * @param relation The relation type
 */
    private void addRelationClicked(
        Person person1,
        Person person2,
        String relation) {
    try {
        if (relation.equals(Relation.CHILD)) {
            person1.addChild(person2);
        } else if (relation.equals(Relation.PARENT)) {
            person2.addChild(person1);
        } else if (relation.equals(Relation.SPOUSE)) {
            person1.addSpouse(person2);
        }
        relationTableModel.personAdded();
    } catch (AddRelationException e) {
        JOptionPane.showMessageDialog(
                this,
                e.getMessage(),
                "Adding relation failed",
                JOptionPane.ERROR_MESSAGE);
    }
}
```

The tenth method evaluated can be seen in Listing 10. The topics selected for further analysis of the method are in Table 23. Half of the informants would have extracted this method to smaller methods. Fourteen informants said that code appeared as unreadable characterizing it for example as *very unintuitive, very unclear,* and *very difficult to read.* Eleven informants pointed out excessive nesting. Four informants suggested a complete refiguring i.e. going thoroughly through the code and rewriting it[13]. No informants said anything that could have been categorized with topic OK.

The need for extract method and the excessive nesting would have been easy to detect with tools. Complete refiguring needed is actually a qualitative comment indicating that the method has become too complex. For detection of poor methods, such as the one in Listing 10, a combination measure of lines of code and cyclomatic complexity could have been used.

[13] As the author of the code I may add that creating this coding horror was not particularly difficult. Maybe it is easier to create poor source code than to understand it.

Listing 10. Method Person.illegalRelation

```
/**
 * Is the new relation illegal
 * Is there allready some kind of relationship between the two people
 * @param relation Relation that holds the people together
 * @return true if the two people already are in
 *                    any relationship with each other
 */
private boolean illegalRelation(Relation relationToBeAdded) {
    //Cannot have relationhips with yourself
    if (relationToBeAdded.getPerson1() == relationToBeAdded.getPerson2())
        return true;
    // Loop through this persons relationships
    for (Iterator iter = vecRelations.iterator(); iter.hasNext();) {
        Relation relation = (Relation) iter.next();
        // We allready have some kind of relation between the persons
        if (relationToBeAdded.equalsPersons(relation))
            return true;
        // Get direct reference to the person this person will have relation
        Person relativeToBe; // The person we will have relation with
        if (relationToBeAdded.getPerson1() == this)
            relativeToBe = relationToBeAdded.getPerson2();
        else if (relationToBeAdded.getPerson2() == this)
            relativeToBe = relationToBeAdded.getPerson1();
        else // this person does not partipate in this relation
            return true; // Should not be reached
        //Rule addition:
        //this person cannot have relation with it's siblings (=sister/brother)
        if (relation instanceof RelationParentChild) {
            RelationParentChild rpc = (RelationParentChild) relation;
            if (rpc.getChild() == this) { //Found Parent
                Person parent = rpc.getParent();
                for (Iterator iter2 = parent.getRelationships().iterator();
                iter2.hasNext();
                ) {
                    Relation parentsRelation = (Relation) iter2.next();
                    if (parentsRelation instanceof RelationParentChild) {
                        RelationParentChild childRelation =
                            (RelationParentChild) parentsRelation;
                        if (parent == childRelation.getParent()) {
                            //Found a child
                            Person child = childRelation.getChild();
                            if (child == relativeToBe) {
                                //Found sibling illegal relation
                                return true;
                            }
                        }
                    }
                }
            }
        }
    }
    return false;
}
```

Table 23. Selected topic frequencies in method FamilyFrame.addRelationClicked

Topic name	Number of informants	Percentage (out of 36)
Extract method	18	50,0%
Readability poor	14	39%
Excessive nesting	11	31%
Complete refiguring	4	11%
OK	0	0%

4.3.3 Summary of the qualitative analysis

Cross case summary of the analyzed methods is in Table 24. In the table, refactoring decision is measured on scale 1-5 with one indicating no need for refactoring and five indicating immediate need for refactoring.

Table 24. Cross-case summaries of the analyzed methods

Method	OK	Refactoring decision:		Summary
		mean	std. dev	
1	7	3,5	1,40	The informants indicated that the method has a long return statement that should be split up (similar to method number 8). Method documentation was not sufficient or was misleading according to the informants. Method also had a large std. dev in the Refactoring decision indicating conflicting opinions.
2	7	3,8	1,26	Method is too long and duplication removal functionality is not needed if different algorithm is used according to informants. Method number 4 is similar to this in the sense that they both operate with the file system and are responsible for applications I/O procedures.
3	22	2,3	1,37	Most informants considered this method OK. Poor naming of called methods and the gender handling are the biggest problems according to the informants.
4	6	3,8	1,28	Method is too long, and the algorithm that reads data from the disk needs improvement according to the informants. Few also pointed out excessive usage of temporary variables.
5	27	1,9	1,18	This is the best method under evaluation according to the amount of the informants' OK comment. There were only few negative comments on minor issues by the informants: some informants would have used other data structure instead of vectors, declaration of parameter should have been done outside of loop.
6	4	4,2	0,96	This is long constructor method that has a poor readability and layout according to informants.
7	21	2,5	1,56	Mostly OK method according to the informants. Although, the method's standard deviation is highest among the methods that indicates conflicting opinions. Most mentioned problems by informants, poor method documentation, layout of parameters, and spelling errors. Only two pointed out that using parameter object would greatly benefit the method.
8	7	4,1	1,15	Method suffers from long return statement, and incorrect layout that hinders readability according to the informants. Some also pointed out that it has duplication with other method in the same class.
9	24	2,2	1,21	Mostly OK method according to informants. Javadoc needs to be updated to tell what the method does. Variable names should be improved.
10	0	4,6	0,60	Suffers from too excessive nesting, is too long, and has a complicated logic according to informants. This is worst method according to number of OK comments.

Summary of the most frequently mentioned topics are in Table 25. The complete set of topic frequencies are in Appendix C. Topics in *italic* are super topics meaning that more than one topic is included in them. From the table we have removed topics that had less than 20 references or were already included in one of the super topics.

The most frequently used topic was OK meaning that the method is fine and there is hardly need for improvement. OK topic had three levels 1, 2, and 3 indicating the level satisfaction expressed towards the method. The second frequent topic was a super topic combining complain and improvement suggestion. Either the method was too long and/or the informant was suggesting that a certain part of the method should be extracted to a method of its own. The third topic was a general comment indicating the method was not readable. The fourth topic, improve comments, was again super topic combining all the complaints and improvement suggestion on the source code comments. The fifth super topic combined the complaint and improvement suggestion on the layout of the code. Sixth topic, change algorithm, indicated that a code can be improved by using a different approach to implement it e.g. if one has coded a sorting routine with bubble sort the code can be improved by using quick sort and if one is using self coded file format to store data it can be replaced by using Java's serialization functionality. At this point, a clever reader has noticed that there is slight difference in the two examples presented above. In the first example, the person programs the improved algorithm by himself / herself and in the second, a pre-built functionality is used. However, both of these were categorized under the change algorithm topic because they both indicate completely new approach in implementing the existing functionality. The seventh super topic combined the complaints of too long statement and the improvement suggestion of splitting up a single statement to several smaller ones. High amount of this super topic could be seen in methods 1 and 8. The eight topic, coding conventions, consisted of minor problems. This meant comments that, by the interpretation of the author, would not have had big impact on the evolvability and were more a matter of opinion e.g. were loop variables declared in the loop or before the loop, for-loop structure instead of while-loop structure, or traversing the vector with elementAt method call rather than using a iterator.

If we compare the most frequently used topics with Figure 9, in page 34, we can see that each *target* from Figure 9 (visual representation, documenting, and structure) will include one or more the most frequent topics. However, three most frequent topics do not fall in to these categories. Topics OK, poor readability, and coding conventions cannot be put in any of the *target groups'* categories. The comments behind topics OK and poor readability are likely formed from a combination effect of the three *target groups* e.g. readability is poor when visual representation is not good, documenting is poor and structure could be slightly better. Often the answers of those topics were also quite general that more specific topic could not be assigned. Finally, coding conventions are a collection of minor issues including problems in all three *target groups*. Thus, this category is created based on the impact of the complain or the improvement suggestion rather than the *target* of it. Often the texts coded with coding convections were also coded to some other category to indicate the *target* of the problem.

Most frequent topics can be split to axes with ends general and specific. Topics OK, poor readability will be the most general where as topics indicating that a method or statement should be split up will be the most specific. Coding convention is also quite general as it only indicates the impact of the issue. Topics indicating poor comments and layout are somewhat general. The topic change algorithm would also appear to be more general than

specific because it may contain several changes. It is good to notice that the topics that appear to be most general could not be placed inside of any single *target group* (visual representation, documentation, structure). With comments that are more specific this grouping can be easily achieved.

Table 25. Most frequent topics

Topics	**Methods**										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Totals**
OK	7	7	22	6	27	4	21	7	24	0	125
extractMethod + longMethod	0	15	2	16	0	22	0	4	0	18	77
readabilityPoor	4	0	1	5	4	18	0	7	1	14	54
commentsImprove	9	4	1	2	2	1	6	3	7	5	40
layoutPoor	1	3	1	4	2	12	2	5	3	4	37
changeAlgorithm	1	8	0	12	3	3	1	3	0	2	33
longStatement + statementSplit	17	0	0	0	2	0	0	12	0	0	31
codingConventions	1	7	1	2	3	0	2	1	2	4	23

4.3.4 Interpretation and summary

In Figure 9 we saw the different topic families that were created based on the answers. The classification of answer *types* to positive, complain, and improvement suggestion is not very interesting for the software engineering community. However, the classification of answer *targets* to visual representation, documenting, and software structure could help researchers distinguish the different sources affecting the evolvability.

By going individually through the evaluated methods we studied the individual characteristics of each method and saw how the informants evaluated them. We saw the specific problems of each method by studying at the coded answers. For example, method number 10 had problems with excessive nesting, but this problem was not mentioned in any of the other methods as one can see from Appendix C. Similar patterns with other topics also appeared where few methods produced the majority of the topics references. Table 25 showed that even the most frequent topics are not uniformly distributed to different methods. This indicates that individual method characteristics play very important part in the evolvability evaluation.

Based on the discussion on automatically detecting the problems (code smells) indicated by the informants, we can say that in some cases automatic detection seemed usable, in others it did not, and there were several cases that fall between these two categories. For example detecting a long method is easily achieved by measuring the method length. However, it is more difficult to measure cases where extract method refactoring should be applied i.e. if individual prefers small utility routines for conditional comparisons. Long statement problem could be automatically detected, but it might require development of some new measures, as we are not currently aware of metrics measuring the statement weight or size.

It is nearly impossible to detect problems such that change algorithm or improve readability. The latter is difficult to detect because it is two vague. Perhaps it could be detected by forming a profile of several methods that suffer from this problem. The former cannot be detected because it requires understanding of the semantics, and knowledge of better solutions and the pre-built functions available in APIs. These functions are not currently possessed by automatic detection tools.

The affect of the proposed refactorings to commonly used source code metrics were also discussed. Generally, the refactorings will cause some metrics to decrease while others increase. For example extract method decreases the size of a single method, but increases the method count of a class. Statement split refactoring will increase method size measured with lines of code, number of statement or cyclomatic complexity, but it will decrease the size or weight of an individual statement. Change algorithm is one of the few refactorings that can decrease some measures without increasing any of the other measures, but as previously discussed, its automatic detection is virtually impossible.

We saw that some of the most frequent topics were very general and that some of them were quite specific. The problem with the general topics is that they hide the interesting details and that that they cannot be effectively categorized. The problem with specific topics is that it is difficult compare them between cases due to the very uneven distribution. Thus, the specific topics greatly reflect the software that was under evaluation. The differences between the specific and general topics are likely to become evident if more studies of this nature are made. It is possible that in the future studies the specific topics can be completely different to the ones we had in here. On the other hand, the general topics and their frequencies should be more or less the same. This way general topics make it easier to perform comparisons cross studies while specific topics increase the existing body of knowledge concerning different problems that may appear in the code.

4.4 Factors Explaining Evaluations – Regression Analysis with Qualitative Data

In this section, we shall return to regression analysis and study whether the topics found in informants answers could be used to explain the refactoring decision. On one hand, this might sound pointless as the informants answers were exactly the rationales for their refactoring decisions. On the other hand, we must acknowledge that if an informant recognized a certain problem, e.g., method is too long, it still does not indicate that the informant would have refactored it. It is possible that the informant did not consider method length to be a very bad problem. Regression analysis makes it possible to compare the effects of topics. For instance we can study which topic families (structure, layout, visual representation), classified as *targets* in Figure 9, have the biggest impact on the refactoring decision.

4.4.1 Regression models

The regression models are in Table 26. We can see that the Simple model, consisting of OK topic in ordinal scale and number of topics in topic family negatives, is able to explain over 70% of the variation. The Target model, consisting of the topics that could grouped to different targets of Figure 9 in page 34, is only able explain little over 40% of the variation. However, the Target&OK model is almost as powerful as the Simple Model

explaining 69,6% of the variation. The Pure Topics model, consisting of all individual topics excluding super topics and OK topics, is only able to explain 37,7% of the variation. The Pure Topics Model has R squared 0,587, but the model suffers when adjusted R square is calculated, because it has 61 topics as the explaining variables Pure Topics model is also performing better when OK topic is added to the model resulting in adjusted R square 0,662. OK topic is part of the original topics, and, therefore, it could be seen as part of the Pure Topics model.

Table 26. Qualitative regression models

Model	Adjusted R Square	p-value
Simple model	0,702	0,000
Target model	0,403	0,000
Pure Topics model	0,377	0,000

Betas and correlations for Simple Model are Table 27. The model is easy to understand. OK topic is a suppressor term for the refactoring decision and the topic family negatives increased the refactoring decision. Both values have high correlation with refactoring decision. OK topic alone is able to predict 66,6% (adjuster R Square) of the refactoring decision, and similarly topic family negatives is able to predict 62,4% (adjuster R Square) of the refactoring decision. Based on the adjusted R square values of OK topic and topic family negatives one might expect the combined model to have higher values than 0,702. To explain this we studied the correlation between the two variables and found out that the variables had significant ($p<0,01$) Spearman correlation minus 0,709. This indicates that the two variables often act simultaneously i.e. if one has high values the other has low values and vice versa. This explains why the Simple Model's adjusted R is not higher.

Table 27. Predictors in the simple model

Topic	Standardized Beta	p-value	Correlation
OK	-0,508	0,000	-0,815
Topic family negatives	0,370	0,000	0,791

Table 28 shows the independent variables of the Target model. From the model, we can see that the topic family Structure, which indicates the number of comments concerning structure, was the most import predictor. Visual Representation also had some effect, while topic family Documenting, which means the number of complaints or improvement suggestion affecting comments or naming, did not have an affect on the refactoring decision. It is not surprising that the refactoring decisions were mostly influenced by the software structure. However, Visual Representation also plays a role in the refactoring decision.

Table 28. Predictors in the target model

Topic	Standardized Beta	p-value	Correlation
Topic family Structure	0,610	0,000	0,608
Topic family Visual Representation	0,197	0,000	0,199
Topic family Documenting	0,035	0,400	0,016

Because Pure Topics Model contained over 60 topics, it is not informative to study them in a one table. However, we can narrate it in text. Three topics had standardized betas over 0,2. They were extract method 0,481, add blank lines 0,237, and long statement 0,210. Especially extract method appears to be strong indicator of the refactoring decision. The problem with pure topics regression model is that there were over 60 predictors and many of the predictors were seldom active, when the methods were evaluated by the informants. Over 40 predictors are active less than ten times, and a regression model that was equally good was created without the variables that were active less than five times. The problem with Pure Topics model is that there really are not enough cases to create a good regression model for over 60 predictors. The Pure Topics model also lacks a measurement of the problem level that the OK topics had as they were classified to three groups. To illustrate this consider the differences in saying "method seems somewhat long" when compared to "method is way too long, and the original programmer should be shot".

4.4.2 Interpretation and summary

From the regression models, we can see that the Simple model performs very well and is able to explain over 70% of the refactoring decision variation. The model has only two independent variables. First, the topic OK measures with three levels the satisfaction the informant feels towards the evaluated software element. Topic family negatives contain the number of different negative comments made on the evaluated software. With these two simple variables, it was possible to explain 70% of the refactoring decision. This indicates that the informants' rationales have real connection with their refactoring decision.

The second regression model called Target Model had three topic families as independent variables. The topic families included the negative comments that could be grouped to one of the groups: structure, documenting, and visual representation. The most important predictor was structure but also the visual representation had some impact while documenting was insignificant predictor. This regression model was only able to explain 40% of the variation. The models weakness was that it lacked all the positive comments of the evaluations. When the OK topic was added to the model, it was almost as good as the Simple model. Additionally, the model lacked the negative comments that could not be put to any of the groups, but this did not have a big impact on the model performance. Based on this data it seems that it is not enough to measure just the negative comments, but one must also take the positive comments in to consideration when trying to predict if certain piece of code will be refactored or not. This idea is very natural since in many cases a piece of software can have good and bad sides, and it seems natural the refactoring decision is based on weighing the both sides.

The Pure topics regression model, lacking all topic families and the OK topic, is able to predict 37,7% of the variation. Although, this model contains some positive topics like "comments are OK" it lacked the most important positive predictor the general OK topic measured in three levels. Adding this to the model greatly increases the models prediction power. We found three variables that had standardized betas over 0,2. Extract method topic had beta 0,478 making it the most important predictor. However, it seems likely that most of the betas can be explained by the evaluated methods i.e. different types of methods would have resulted in different topics getting high betas. However, the beta in extract method is clearly higher than the rest. Thus, it could indicate that this topic would be important predictor in studies containing different methods as well. It is also interesting that extract method performs much better than general negative comment *readability poor* although they both have more than 50 mentions. It could be that when informant indicates that a new sub method should be extracted he/she is much more certain about the refactoring decision than when stating rather vague complain about the readability. This would indicate that the decision whether to refactor would be affected by the informants' knowledge on how to make refactorings.

5 Discussion

This chapter recaptures the work done to answer the research questions, summarizes the answers to the research questions, discusses the limitations of the study, and finally provides ideas for further research.

5.1 Answering the Research Questions

This section will answer the research questions from Section 3.1 based on the results that were presented in Chapter 4. We will also make comparisons to prior work.

5.1.1 Research question 1

Is there an interrater agreement in subjective evolvability evaluation?

This research question was studied in Section 4.1. Kendall's coefficient of concordance, also known as Kendall's *W*, was used to measure the agreement between raters.

In the evolvability evaluations, we saw that code smells Long Method and Long Parameter List produced a high agreement between raters having Kendall's *W*s of 0,777 and 0,816 respectively. The high agreement on these smells is not surprising, since both of them should be easy to evaluate and rank. Long Parameter List can be clearly seen by looking at how many parameters are passed to the method. Long Method could be little more difficult to recognize since the definition given told that such methods have low cohesion, are difficult to understand, and reuse. Still the informants had very high agreement on the Long Method smell, as well.

The Feature Envy smell had the lowest coefficient of concordance with 0,238. However, from the feedback of the experiment it was learned that some informants (3/46) felt that they did not completely understand what was meant by Feature Envy smell, although this was not specifically asked. This can partly explain the low interrater agreement on this case. However, the two persons that were used to pre-validate Experiment A did not make such comments. Therefore, there was no possibility to fix this problem beforehand. The fact that two persons used to pre-validate the experiment did not indicate that the lack of understanding in Feature Envy smell suggests that this problem may not have affected all of the informants.

The Kendall's *W* for the refactoring decision in Experiment A was 0,353 and in Experiment B 0,397. This was considerably lower than on code smells Long Method and Long Parameter List. *W* values in the refactoring question in both experiments were very close to each other (only a difference of 0,044). This indicates that the level of interrater agreement is not affected by the different setups in the experiments. However, one might expect the agreement on the refactoring decision to be higher in Experiment A where the informants had the smell descriptions available to help them in making the refactoring decision.

The refactoring question really was at the heart of the experiments because it asked if the method is in such a bad shape that it should be improved to make it more evolvable. It seemed unlikely for any misunderstanding concerning this question. Hence, the result seems to indicate that there are differences in people's opinions on if a certain piece of code should be refactored.

Based on the data the answer to the research question 1 is two-folded. For simple code smells Long Method and Long Parameter List there was a high agreement between the raters. For the refactoring decision and the Feature Envy smell, the level of agreement was considerably lower. Since all evaluations are significant, we must conclude that there is partial concordance among the raters in all evaluations. However, the level of agreement is not satisfactory in all cases.

Comparison to prior work (Coleman et al. 1994; Coleman et al. 1995; Kafura and Reddy 1987; Kataoka et al. 2002; Shepperd 1990; Shneiderman 1980; Welker et al. 1997) is challenging due to: lack of proper representation of the evaluation data (Coleman et al. 1994; Coleman et al. 1995; Oman and Hagemeister 1994; Shepperd 1990; Welker et al. 1997), lack of statistical power (Kataoka et al. 2002; Mäntylä et al. 2004), use of non-standard statistical methods[14] (Mäntylä et al. 2004; Shepperd 1990; Shneiderman 1980). All the prior work lacked calculation of statistical significance on the interrater agreement and the Kendall's *W*, which makes it impossible to say whether the raters really had agreement on the software evaluated or not. To make things even worse, the data is not available to public except in the Shneiderman's study (Shneiderman 1980), who has included some of the raw data in his book. Unfortunately, even in Shneiderman's study, which could be excellent comparison point for this study, the data is reported in a way that makes it impossible to track an individual evaluator's answers through the programs she had evaluated. Therefore, we cannot compare our results with any prior work.

5.1.2 Research question 2

How much of the evolvability evaluation of a software element can be explained by the software element and the informant?

This research question was studied in Section 4.2. The factors affecting the evolvability evaluations were searched through the measurable characteristics of a software element (source code metrics) and the demographic data of an informant. This research question was studied using categorical regression founded on optimal scaling, which makes it possible to use continuous and non-continuous variables as both dependent and independent variables.

We saw that the evaluations on code smells Long Method and Long Parameter List could be predicted with good accuracy by the regression models. In Long Method smell 71,2% of the evaluations could be explained by the regression model consisting of source code metrics. As expected, the lines of code metric was the most important predictor for the Long Method evaluations. However, the Metric model without the lines of code metric also explained 61,8% of the evaluations. This is caused by the correlation the source code metrics have with each other. In Long Parameter List smell evaluations the Metric model explained 76,1% of the evaluations. The most important predictor in the model was the number of parameters, and unlike in the Long Method evaluations there was no substitute for this predictor.

The explanation power of the source code metric based regression models diminished when we studied the refactoring decision and the Feature Envy code smell evaluations. The

[14] The researchers have calculated the percentage of answers that were off by n steps in ordinal scale, or they calculated averages and standard deviations from the ordinal scale. In such procedure, no statistical significance can be calculated.

source code metrics explained only 9,8% of the Feature Envy smell evaluations. For the refactoring decision, the percentage of the evaluations explained by the Metric models was 31,9% in Experiment A and 26,1% in Experiment B. This number was lower than what one would expect. In these experiments, the source code metrics also suffered from high collinearity, which makes it difficult or nearly impossible to say how good predictor each of the metric was. Based on the experiments we can only say that a line of code metric appears to be the best refactoring decision predictor. The prediction power of the rest of the metrics cannot be ranked because of having small beta values, having very different beta values in experiments A and B, or having negative beta values when having positive correlation with the refactoring decision while also having strong correlation with the best predictor and therefore suffering from multicollinearity. For interesting empirical study on C&K source code metrics (Chidamber and Kemerer 1994) multicollinearity look for (Succi et al. 2005).

In Experiment A we were able to use the smell evaluations of each informant to create regression model that explained 61,8% of the refactoring decision variance. This result is not surprising because the smell evaluations and the refactoring decisions were subjective evaluations that should have connection with each other. The experiment set up could have affected this as well: first, the informants were told that smells are bad, then they were asked to evaluate the existence of the smells, and finally they were asked whether they would refactor the method to remove the smells in order to keep the software evolvable. Thus, we could expect the regression model with the smells to explain even greater deal of the variation.

We also studied the demographic variables as predictors for a refactoring decision, but their explanatory power was low. We even tried to improve the gathering of demographic data in Experiment B grounded on Experiment A, but still the background variables had only minor explanatory power. In fact, the demographic variables performed slightly better in Experiment A, but this was affected by having more demographics variables in Experiment A rather than a real improvement in the data variables.

Comparing these results to the results of research question 1 reveals us that both the interrater agreement and the explanatory power of metric regression model have similar two-folded structure. Both perform well on Long Method and Long Parameter List smell evaluations. Similarly, both the inter-agreement and the regression models have low values when it comes to refactoring decision and especially to Feature Envy evaluations. Clearly, there is a connection between these two, and it is caused by the fact that the source code metrics of any method will remain the same even if there is disagreement between raters. Thus, if there is a disagreement whether a certain method should be refactored, it automatically means that the code metrics of that method cannot make up a strong regression model that would predict the refactoring decision because there is a disagreement on the issue. Naturally, this does not affect the regression model created from the smell evaluations because the possible disagreement in the refactoring decision is likely to be reflected in the smell evaluations.

Comparison to prior studies is not clear-cut because they have not utilized regression analysis. Regardless, we can try to make some comparisons. Kafura and Reddy (Kafura and Reddy 1987) concluded that the expert evaluations on maintainability were in conformance with the complexity source code metric. This conflicts with our results since we showed that metrics were not effective predictors of the refactoring decision i.e. the evolvability improvement need. However, their results are based on interviews on software maintainability while we used the refactoring decisions on an ordinal scale survey. Software

Engineering Test Lab at the University of Idaho used subjective evaluations to create a metrics based maintainability measure (Coleman et al. 1995; Welker et al. 1997). It is therefore quite natural that their metric correlated well with subjective evaluations.

Our prior work (Mäntylä et al. 2004), which studied code smell evaluations at the module level on an industrial setting, concluded that source code metrics and code smell evaluations did not correlate. However, this study shows that simple code smells and source code metrics have a relationship. The difference is likely to stem from three limitations in our prior study. First in our prior work, we used higher (module) level evaluations. Second, the evaluations were based on recollection. Third, the evaluators had been working with the software modules and, thus, had a more personal bond with the software under evaluation. We think that these dissimilarities can explain the different results.

5.1.3 Research question 3a

What factors act as the rationales for the refactoring decision, could these factors be automatically detected, and what would be the effect of the improvement suggestion factors to common source code measures?

This research question was studied in Section 4.3 by exploring the qualitative data provided by the informants of Experiment B. The informants were required to give rationales for their refactoring decision. These rationales, which consisted of one to ten lines of text, were analyzed with the coding paradigm by using atlas/ti software.

The rationales were grouped to one or more of the following three topic t*ypes*. The informant could praise, complain, or suggest improvements to the software element. Often complain was accompanied by an improvement suggestion and vice versa. It was evident that complains and improvement suggestions had a relationship with each other. The complain was the cause for the improvement suggestion e.g. a rational might have complain pointing out that the method is too long and it would additionally contain suggestion splitting the method into sub-methods. This is hardly surprising and similar complaint-improvement pairs can be found for example in Fowler's work (Fowler 2000) where we can see Code Smells as the complaints and the appropriate refactorings as the improvements. The praises given by the informants indicated that the method was already in a good shape, and therefore they are opposite of the complaints and improvement suggestions. Thus, it is possible to further classify the answers to two types one containing the positive comments and other containing the negatives.

In addition to their *types*, the rationales were also classified based on their *targets*. Three *target groups*, namely Visual Representation, Documentation, and Structure were identified. Visual representation means issues that make a method easy for a human eye to study, indentation, blank line usage, etc. Documentation contains all the (textual) information within the source code that makes the programs easier to comprehend, e.g. naming of variables and comments. Structure stands for the source code composition that is eventually parsed by the compiler to a syntax tree. Structure is clearly distinguishable from documentation and visual representation, because the latter two have no impact on the program run-time operations or the syntax tree generated from the source code.

Not all the answers could be classified to any *target group* because they were too general e.g., comments such as "method is difficult to understand" cannot be classified to any of the three *target groups*. Additionally, many positive comments were given that were very general. They were classified to three OK groups based on how positive the comment appeared to be.

It is difficult to provide a brief overview of the detailed topics that that fall inside of the *target groups* discussed above. However, here we will try to summarize the most interesting points. Firstly, over 60 different topics, given as the rationale for the refactoring decisions, were discovered. This number indicates that the informants were able to discover wide range of different issues in the evaluated software. Secondly, many topics were method specific, meaning that there could be several mentions of a single topic in one method and only few or not any in the rest of the methods. Thirdly, refactoring change algorithm, which means implementing some functionally with completely different approach (e.g. use Java's serialization functionality to save data rather than implementing own file format), appeared frequently in the answers. We feel that it could offer one of the greatest potentials to make code more evolvable. Fourthly, hardly any answers pointed out the excessive number of parameters in the methods where such comment would have been appropriate. This is interesting because in answer to research question one, in Section 5.1.1, we saw that Long Parameter List code smell evaluations had the highest agreement among the informants. Still only few informants mentioned this as a problem to be fixed in Experiment B. Likewise, too excessive coupling from the method, in Listing 1, to another class, and the fact that actually this method should have been moved to inside of the class it was excessively coupled with, was only mentioned by two of 36 informants. These results indicate that the given evaluation criterion can greatly affect the issues found in the evaluated code. Fifthly, there were conflicting opinions in the informants rationales, for example in the method, in Listing 6, an informant claimed that the method was nicely grouped, while another informant claimed just the opposite saying that the method was poorly grouped. These kind of conflicts found in the qualitative data provide insight to the cases where there is low interrater agreement.

Automatic detection of some topics is possible while in others it is not. In general, it easy to measure method length and then use a threshold to determine whether the method is too long. It is more difficult to discover whether certain piece of method should extract to a method of it own because preferences vary greatly, e.g. some people for instance may prefer small utility methods for Boolean expressions. In some cases, automatic detection would also require new (according to our knowledge) specialized metrics like statement size or weight metric. Change algorithm refactoring was recognized as important, but currently its detection does not seem feasible. Detection of general topics seems hard to do in practice. Maybe this could be done by imputing several (tens or preferable hundreds of) methods' to a machine learning system that could then generate a profile of what is meant by the general topic. However, even such a machine learning system would require agreement between raters in order it to work correctly.

The effects of the suggested refactorings to common source code metrics are in some cases trade-offs. Splitting long statement will result in more statements and increased cyclomatic complexity, but it will reduce the weight of individual statement. The move method refactoring on the other hand will not cause tradeoffs in the metrics, as it will ideally just reduce couplings leaving every other metrics as they were. Change algorithm is another refactorings detected that can decrease some measures without increasing any of the others. Unfortunately, it seems that the refactorings' effects to source code metrics cannot be better summarized based on this study.

Siy and Votta (Siy and Votta 2001) studied data of 130 code inspection sessions. They focused on "soft maintenance-issues" that are similar to the rationales analysed in our

work. They also categorized part[15] of their qualitative data based on the goals the issue tried to resolve. They came up with four groups, namely Documentation, Style, Portability and Safety. Documentation contained 47% of the issues while style had 46% of the issues. Thus, combined these two groups were responsible for 93% of the issues.

Siy and Votta further categorized the documentation group to sub-groups called Clarification, Correction, and Documentation of Future work. Similarly, this work has the *target group* Documenting. However, our definition is larger as we also considered the naming of source code elements (e.g. variables and routines) to be part of Documenting, where as Siy and Votta only consider the comments in the code. In our data, we had no topics that could be categorized under the Documentation of Future work in grouping by Siy and Votta. Perhaps this is because our informants were not the developers of the application, and thus their mind set might not been so future development oriented. This work has only one topic that could be place under the Correction group by Siy and Votta. This is topic commentsPoor. However, our commentsPoor topic also contained few answers where topics were not incorrect, but just poor, or incomplete. Rest of our topics referring to the comments would be placed under the group Clarification by Siy and Votta.

Siy and Votta further categorized the Style group to sub-groups called Clean-up, Renaming, Debugging, and Cosmetic. The Clean-up group has perfect match in this work's *target group* Structure. The Renaming group means the naming of source code elements (e.g. variables and routines). In this work renaming was included in the *target group* Documenting. The Debugging group has no match in this work. The Debugging contained suggestion of adding debugging and trace statements to the source code. Finally, the cosmetic group has a perfect match in this work with *target group* Visual Representation.

The above comparison between this work and the study of Siy and Votta shows many similarities in the classification. However, the classifications also have differences. Siy and Votta had groups Documentation (with sub-groups Clarification, Correction, and Documentation of Future work), Style (with sub-groups Clean-up, Renaming, Debugging, Cosmetic), Portability, Safety. This work had *target groups* Structure, Documentation, and Visual Representation. The differences in the groupings are probably caused by the different environments in which the data was gathered. Therefore, it is pointless to argue why one grouping would be superior to the other. Maybe in the future there will grouping that will take best sides of both groupings.

According to our knowledge other similar prior work of this kind has not been performed.

5.1.4 Research question 3b

Can the factors of the rationales predict the refactoring decision, and, if yes what are the most important factors?

This research question was studied in Section 4.4. Similar to research question 2 this was studied by using categorical regression based on optimal scaling. The predictors were the coded answers given by the informants in their refactoring decision rationales.

Three regression models Simple model, Target model, and Pure Topics models were created. Simple model consisted of two predictors: positive OK topics and the topic family

[15] The source (Siy and Votta 2001) does not reveal how many code inspection session's data was used to create the grouping

Negatives, which was created by summing up all the different negative comments made in each answer. Target model was created from all the negative comments that we were able to categorize under code families Visual Representation, Documentation, or Structure. Finally, Pure Topics model contained all the topics identified during the coding process. Simple model was able to explain 70,2% of the refactoring decision variation. Both of its predictors were important with std. betas -0,508 and 0,370 for OK topics and topic family Negatives respectively. Target model was able to explain 40,3% of the variation. Topic family Structure was the most important predictor with std. beta of 0,610 while topic family Visual Representation had std beta of 0,197. Topic family Documenting was not significant predictor in the model. Pure Topics model was able to explain 37,7% of the variation. In Pure Topics model there were over 60 predictors, but only three of them had std betas over 0,2. Those topics were extract method, add blank lines, and long statement with betas 0,481, 0,237, and 0,210 respectively. Target model and pure codes model were both created without the OK topics. If the results of this positive OK topic were added to these models, their predictive power increased considerably, but not to the level of the simple model.

Based on the results we can conclude that the factors discovered in the qualitative answers can be used to predict the refactoring decision. It appears that for successful regression model we must measure both the positive and negative comments discovered in the answers. The most important positive predictor is the OK comment that indicates the amount of positive comments given to the method in three-point ordinal scale. There were several negative comments. The best predictor out of the negative comments was created by summing up all the negative comments in to a topic family called negatives. When studying the negative comments with the Target model we saw that code Structure is the most important predictor while Visual Representation also had some impact. Studying this further with pure codes model revealed that suggestions to extract method and complain concerning long statement were most important structural predictors. From topic family Layout add blank lines rose as the most important predictor.

Comparison to the answers of research questions 2 in Section 5.1.2 shows that the qualitative elements provided by the informants are far better refactoring decision predictors than the source code metrics or the demographics. Now we can compare the prediction power of the qualitative elements gathered in experiments A and B. In Experiment A, the qualitative elements were gathered by asking the informants about three code smells. In Experiment B, the qualitative elements were gathered from the refactoring decision rationales. The qualitative regression model in Experiment A, i.e. the Smell model, was able to explain 61,8% of the refactoring decision variation. The qualitative regression model, i.e. simple model, in Experiment B performed slightly better explaining 70,2% of the variation. Maybe a better prediction power could have been achieved in Experiment A if opinions on more code problems would have been asked.

We are not aware of any studied were qualitative answers concerning code evolvability would have been used as a basis for regression analysis. Therefore, we are not able to perform any comparison on prior work.

5.2 Limitations

This section assesses the limitations of the study. Threats to internal validity and external validity are studied as well as some special considerations on experimental design. Finally, we will propose some improvements to the experiments. Threats to validity were identified

based on Campbell and Stanley(1966), and Coolcan (Coolican 1999) who cites Cook and Campbell (1979). The special considerations to experiment design are done based on Juristo and Moreno (2001).

5.2.1 Threats to internal validity

In Experiment B, the reliability of the respondents' procedures was questionable since it was done as a web survey, and there was no control over the respondents. However, a similar situation could have occurred in Experiment A, because we had no means to make sure that the informants actually paid attention to the introduction lecture they were given prior to the experiment.

Experiment B lacked the randomization of the methods evaluated. The evaluation order of the methods could have caused bias to the evaluation results. In Experiment B, the informants had the possibility to go back and forth in their answers if they wanted to check or change something in their prior answers. This should have limited the effect cause by the lack of randomization.

In both experiments, collection of demographics could have been improved. In Experiment A, we did not collect any information of the students' success in prior programming or software engineering courses. In Experiment B, we collected the students grades on selected programming courses, but even this could have been improved by allowing students to list other relevant programming or software engineering course that were not specifically asked for. However, we feel that this limitation in Experiment B would not have had big impact on the prediction power of demographics based regression models.

5.2.2 Threats to external validity

There could be interaction between the selection of evaluators and the issue studied in the experiments. In both experiments, the population was the students participating in the course. However, proper sampling was not done to select the individuals who would become the evaluators of the experiment. Instead, the informants were those interested in receiving the extra credit for their course grade. This sampling method could have caused bias. It is possible that students who were more interested in this topic participated in the experiment. Based on the course grades there actually was a slight bias in both experiments towards better performing students.

Another threat to external validity comes from the population. Generalizing the results obtained using students to developers in industry might not be possible. We may argue that students are a too homogenous group, and, therefore, the results are too good. Furthermore, one may argue that as the evaluators had a varying amount of industrial programming experience (from zero to fifteen years), the population is too heterogeneous. However, also teams of industrial developers can have fluctuating levels of homogeneity.

The selection of the evaluated software elements is another threat to external validity. It is possible that with a different set of software elements different results could be obtained. This is particularly crucial for the individual topics mentioned in the refactoring decision rationales that were studied in Section 4.3. Therefore, this mainly concerns answer to researcher question 3a

The source code metrics used to measure the evaluated methods were limited to six different metrics, which were introduced in Section 3.2.5. With a different set of metrics, the results could have been different. However, it must be pointed out that the goal was

not to discover the best metrics to predict refactoring decisions, but to test how a few widely used measures perform in predicting the evaluations.

It is possible that the results represent more the effect of the experimental setting than what it would be in the real world. One may argue that in the real world the agreement between raters would be better since the raters would understand and study the evaluated piece of software longer and more thoroughly.

Finally, the qualitative data and the results are sensitive to researcher bias, i.e. a researcher will discover whatever the researcher thought he/she should discover. Therefore, different researcher might have discovered different topics and topic families to be important in the refactoring rationales. This limitation cannot be efficiently avoided in qualitative studies.

5.2.3 Considerations of experimental design

Juristo and Moreno (Juristo and Moreno 2001) list consideration for experimental design in software engineering. This section will explain how these considerations were addressed in the experiments.

Learning effect means that informants apply the technique differently after they have learned to use it.

In Experiment, A this could have caused differences in the evaluations of the methods depending on the order of the methods. To prevent this in Experiment A, the order of methods was randomized, so that no two informants evaluated the methods in the same order. This prevents the learning effect from biasing the results.

In Experiment B, no randomization was performed because the web-based system lacked such feature. In Experiment B no pre-determined refactoring criteria were given. This means that there was less material to learn from. Finally, in Experiment B the informants had the possibility to go back to their previous answers if they later changed they mind. However, it must be concluded that the lack of randomization makes Experiment B weaker compared to Experiment A when evaluating the learning effect.

Boredom effect means that informants get tired or bored during the experiment, which will lead in poorer performance due to the end of the experiment. As with the learning effect, the randomization of the methods can partly block this effect from the results.

Experiment A lasted only for about hour and half. The informants were motivated with extra-point given to them if properly finishing the experiment. It is possible that informants got bored since they had five minutes to evaluate each method. For the shorter methods this was clearly too long. However, even if the informants got bored when waiting for the permission to move to the next method, this extra time was necessary in order to make sure that the evaluation was not performed too quickly. Performing the evaluation too quickly can happen when informants are bored and want to complete the experiment quickly. Although the five minute evaluation time caused boredom, we believe it also reduced the haste effect and thus affected positively on the evaluation quality.

In Experiment B, the informants were allowed to evaluate the methods and provide rationales in their own pace. Only the final result was controlled in grading the evaluations and making sure no one got any points without actually studying the methods. As there were only ten methods to be evaluated and informants were allowed to proceed at their own pace it seems unlikely that boredom might have biased the evaluations. However, it is possible that some informants got bored and proceeded more rapidly than they should

have. Still, we believe that by grading the rationales we prevented most of the haste that might have occurred through boredom effect. Experiment B seems to be stronger than Experiment A when evaluating against the boredom effect.

Enthusiasm effect concerns mostly cases where two different techniques are applied. In this type of scenario, it is possible that the informants using the new technique are ecstatic while the informants using the old technique are less motivated. However, our experiments did not involve a comparison of two techniques this effect is not an issue.

Experience effect deals with the case where informants are more experienced with the traditional technique and thus perform better. Again, this effect cannot occur in this experiment.

Unconscious formalization means that informants learn from the first technique and unconsciously apply it to the second technique. Again, this effect cannot occur in this experiment.

Assurance concerning the procedure implemented by the subjects raises the issue of being sure that the informants actually used the technique as instructed.

In Experiment A, there was no particular technique for locating the smells or evaluating the refactoring need. The students were given the description of the smells, and the UML description of the application. Additionally, an application demonstration was shown. During Experiment A two instructors were present who made sure that the students actually looked at the methods and did not proceed to the next method until given permission to do so. Therefore, the setting of Experiment A was quite controlled. The informants were also told that they would not receive the additional points if they did not evaluate the methods according to the smell descriptions. Still we cannot ever be sure that the students actually did as they were told. However, in this setting it seems likely that at least most of the students did as they were told.

In Experiment B, there was no control of over the subjects. However, there was no technique for making the refactoring decision or finding suitable arguments to support their decision. The only control came through the subject refactoring decision rationales. Experiment A had better control over the individuals. However, Experiment B had better control of the answers as written output required.

Setting effect deals with the emotional state of the informants. This can occur when experiment runs over several days. This effect means that it is not advisable to have two day experiment on Thursday and Friday, because the informants' emotional state will be different in Friday because of the upcoming weekend. Experiment A only lasted for two hours in Monday afternoon with no holidays in sight. Experiment B was performed as a web-survey and therefore the student might have performed in different days with different emotional state.

Table 29. Summary of the experiment design consideration in the experiments

Consideration	Applicable, prevention	
	Experiment A	**Experiment B**
Learning effect	Partly. Randomization	Partly. There was no prevention, but there was no particular technique utilized either.
Boredom	Yes. Randomization and Forced time to spent on evaluation to avoid hastily evaluations	Yes. Grading of the rationales provided by the informants to prevent hastily evaluations.
Enthusiasm effect	Not Applicable	Not Applicable
Experience effect	Not Applicable	Not Applicable
Unconscious formalization	Not Applicable	Not Applicable
Assurance concerning the procedure implemented by the subjects	Partly, Instructors present during the whole time of the experiment, controlled time for each evaluation, thereat to take additional points away if not evaluating as instructed	Partly. No control over the subject. No procedure given. We controlled the quality of the answers.
Setting effect	No, performed on Monday night with no holidays in sight.	Yes. We had no control over subjects. Therefore, they might have performed the experiment on any day.

5.2.4 Improvements of the experiments empirical design

This section can be valuable to those who wish to study this area further.

1. The selection of the evaluators should be random.
2. The population should be more diverse.
3. More software elements should be used for evaluation to reduce the possible bias.
4. More source code metrics not suffering from collinearity[16] should be used.
5. More code smells should be studied to allow better comparison between the evaluations.
6. There should be several questions describing each smell to allow assessing reliability of answers.
7. A pre-exam should be held to see how well the evaluators understand the issues to be evaluated.

[16] Collinearity means a linear or near linear relationship among the independent variables of a regression model.

6 Conclusions and Future Work

The purpose of this work was to study the subjective software evolvability evaluation, i.e. evaluation of the existence of certain code problems called code smells and the refactoring decision. First, we suggested the use of the term software evolvability over the traditional term software maintainability. To position our research, we provided four viewpoints to software evolvability. Based on these viewpoints, we focused our empirical study on the subjective evolvability evaluation. The empirical research was carried out with two student experiments where we assessed the interrater agreement and the factors explaining the evolvability evaluations.

We have seen that the interrater agreement is high for the simple code smells Long Method and Long Parameter List, but considerably lower for the refactoring decision and the Feature Envy code smell. Regression models based on source code metrics explained over 70% of the evaluations of Long Method and Long Parameter List smells, but explained only about 30% of the refactoring decision. The best predictors of the refactoring decision were the rationales of the refactoring decision explaining over 70% of the variation and the evaluations of the code smells explaining more than 60% of the decisions. It is hardly surprising that the subjective evaluation of smells and the rationales for the refactoring decision were the best predictors. Our regression analysis of rationales also revealed, that the positive comments were good predictors of the lack of refactoring need. They were better at predicting the lack of refactoring need than the negative comments were at predicting the refactoring need.

Analysis of the qualitative data, i.e. refactoring decision rationales, revealed that the method under study greatly affect the contents of the rationales. We also proposed categorization of the negative comments and improvement suggestion to three groups, namely Structure, Documenting, and Visual Representation. By studying the qualitative data of Experiment B and comparing it to the smells evaluations of Experiment A we found that the given evaluation criteria greatly effects the evaluation results. For example, Long Parameter List was identified often in Experiment A and yet it was seldom mentioned in the refactoring decision rationales of Experiment B even though the evaluated code was identical. Some of our qualitative findings confirm that evaluators can have very conflicting opinions of the same method. Based on the qualitative data we also considered the automatic detection of code problems and improvement suggestion. We found out that some code problems would need new code metrics, e.g. statement weight, and that some code problems could not be detected or measured at all. Finally, when considering the effects of the improvement suggestion to code metrics, we found that most refactorings would decrease some metrics while increasing others.

High interrater agreement on the simple code smells implies reliability of the smell evaluations. Thus, there should be no need to double-check the evaluations of those smells by tools or another developer. Additionally, for the simple code smells the prediction by the code metrics based regression model was also quite accurate. This suggests code metrics tools usage as an effective approach in highlighting straightforward problems in the code.

Lower interrater agreement of the refactoring decision indicates a possible unreliability in the developers' evaluations. To compensate this it seems advisable to double check the evaluation at least from time to time because false judgments may lead to poorly evolvable software.

The results indicate that it might not yet be feasible to build tool support simulating real-life subjective refactoring decisions. This is based on three findings. First, the refactoring decisions had low interrater agreement. Second, the metrics-based regression models were only able to explain 30% of the refactoring decisions. Third, some refactoring decision rationales were difficult or impossible to measure and detect, or would require new metrics to make the detection possible.

The success of subjective and qualitative data in predicting the refactoring decision is not surprising. Nevertheless, they are the best predictors for the refactoring decision. Although, there are reliability issues with subjective and qualitative evaluation, this study has shown that subjective evaluation can be reliable if given criteria is simple enough, e.g., code smell Long Parameter List. Therefore, we are planning to make more studies with more diverse code set and with industrial developers to study the qualitative elements of software evolvability in more detail. Hopefully, this way we can come up with criteria that will achieve high interrater agreement and be applicable in real world code evaluation cases. This type of criteria can be used to train the developers in making code that is more evolvable. Qualitative data can also suggest new code metrics, i.e. which aspects of the source code are actually worth measuring. Understanding qualitative aspects of software evolvability would also enable us to build better tool support for the refactoring decisions.

References

AFOTEC. 1996. *Software maintainability evaluation guide*. DEPARTMENT OF THE AIR FORCE, HQ Air Force Operational Test and Evaluation Center.

Arnold, R.S. 1989. Software restructuring. *Proceedings of the IEEE* 77: 607-617.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., and Kontogiannis, K. 2000. Advanced clone-analysis to support object-oriented system refactoring. Proceedings of Seventh Working Conference on Reverse Engineering, , 98-107.

Bandi, R.K., Vaishnavi, V.K., and Turk, D.E. 2003. Predicting maintenance performance using object-oriented design complexity metrics. *Software Engineering, IEEE Transactions on* 29: 77-87.

Bansiya, J., and David, C.G. 2002. A hierarchical model for object-oriented design quality. *Software Engineering, IEEE Transactions on* 28: 4-17.

Briand, L.C., Daly, J.W., and Wüst, J.K. 1997. A unified framework for cohesion measurement in object-oriented systems. Proceedings of the Fourth International Software Metrics Symposium, , 43-53.

———. 1999. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on* 25: 91-121.

Brown, W. J., R. Malveau C., H. W. McCormick, and T. Mowbray J. 1998. *AntiPatterns: Refactoring software, architectures, and projects in crisis*. New York: Wiley.

Campbell, D. T., and J. C. Stanley. 1966. *Experimental and quasi-experimental design for research*. Chicago, USA: Rand Mcnally College Publishing Company.

Chidamber, S.R., Darcy, D.P., and Kemerer, C.F. 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. *Software Engineering, IEEE Transactions on* 24: 629-639.

Chidamber, S.R., and Kemerer, C.F. 1994. A metric suite for object oriented design. *Software Engineering, IEEE Transactions on* 20: 476-493.

Chikofsky, E.,J., and Cross, J.,H. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7: 13-17.

Coleman, D., Lowther, B., and Oman, P.W. 1995. The application of software maintainability models in industrial software systems. *Journal of Systems and Software* 29: 3-16.

Coleman, D., Ash, D., Lowther, B., and Oman, P.W. 1994. Using metrics to evaluate software system maintainability. *Computer* 27: 44-49.

Cook, T. D., and D. T. Campbell. 1979. *Quasi-experimentation: Design and analysis issues for field settings.* Chicago, USA: Rand Mcnally College Publishing Company.

Coolican, H. 1999. *Research methods and statistics in psychology.* 3rd ed. London, United Kingdom: Hodder & Stoughton.

Cusumano, M. A., and R. W. Selby. 1995. *Microsoft secrets.* USA: The Free Press.

Ducasse, S., Rieger, M., and Demeyer, S. 1999. A language independent approach for detecting duplicated code. Proceedings of the International Conference on Software Maintenance, Oxford, England, UK, 109-118.

Fowler, M. 2000. *Refactoring: Improving the design of existing code.* 1st ed. Boston: Addison-Wesley.

Fowler, M., and K. Beck. 2000. Bad smells in code. In *Refactoring: Improving the design of existing code.* 1st ed. , 75-88. Boston: Addison-Wesley.

Genero, M., Piatini, M., and Manso, E. 2004. Finding "early" indicators of UML class diagrams understandability and modifiability. Proceedings of International Symposium on Empirical Software Engineering, , 207-216.

Genero, M., Piattini, M., and Calero, C. 2002. Empirical validation of class diagram metrics. Proceedings of the International Symposium on Empirical Software Engineering, , 195-203.

Grady, R.B. 1994. Successfully applying software metrics. *Computer* 27: 18-25.

Halstead, M.,H. 1977. *Elements of software science.* New York: Elsevier.

Harrison, R., Counsell, S.J., and Nithi, R.V. 1998. An evaluation of the MOOD set of object-oriented software metrics. *Software Engineering, IEEE Transactions on* 24: 491-496.

Henderson-Sellers, B. 1996. *Object-oriented metrics.* Upper Saddle River, New Jersey, USA: Prentice Hall.

Hitz, M., and Montazeri, B. 1996. Chidamber and kemerer's metrics suite: A measurement theory perspective. *Software Engineering, IEEE Transactions on* 22: 267-271.

IEEE. 1990. *IEEE standard glossary of software engineering terminology.* New York: The Institute of Electrical and Electronics Engineers, Inc.

———. 1998. *IEEE standard for software maintenance*. New York: The Institute of Electrical and Electronics Engineers, Inc.

Juristo, N., and A. M. Moreno. 2001. Specific considerations for experimental designs in software engineering. In *Basics of software engineering experimentation*. 1st ed. , 116-119. Boston: Kluwer Academic Publishers.

Kafura, D.G., and Reddy, G.R. 1987. The use of software complexity metrics in software maintenance. *Software Engineering, IEEE Transactions on* 13: 335-343.

Kataoka, Y., Ernst, M.D., Griswold, W.G., and Notkin, D. 2001. Automated support for program refactoring using invariants. Proceedings of International Conference on Software Maintenance, Florence, Italy, 736-743.

Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. 2002. A quantative evaluation of maintainability enhancement by refactoring. Proceedings of the International Conference on Software Maintenance, Montreal, Canada, 576-585.

Kendall, M., Sir. 1948. The problem of *m* ranking. In *Rank correlation methods*. 5th ed. , 117-143. London: Edward Arnold.

Legendre, P. 2005. Species associations: The kendall coefficient of concordance revisited. *Journal of Agricultural, Biological and Environmental Statistics* 10: 226-245.

Lehman, M.M. 1980. On understanding laws, evolution, and conservation in the large-program life cycle. *The Journal of Systems and Software* 1: 213-221.

Li, W., and Henry, S.M. 1993. Object-oriented metrics that predict maintainability. *Journal of Systems and Software* 23: 111-122.

Lorenz, M., and J. Kidd. 1994. *Object-oriented software metrics*. Upper Saddle River, New Jersey, USA: Prentice Hall.

Mäntylä, Mika V. 2003. Master's thesis: Bad smells in software - a taxonomy and an empirical study.

Mäntylä, M.V., Vanhanen, J., and Lassenius, C. 2003. A taxonomy and an initial empirical study of bad smells in code. Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 381-384.

———. 2004. Bad smells - humans as code critics. Proceedings.20th IEEE International Conference on Software Maintenance, 2004. Chicago, Illinois, USA, 399-408.

Maruyama, K., and Shima, K. 1999. Automatic method refactoring using weighted dependence graphs. Proceedings of the International Conference on Software Engineering, Los Angeles, CA, USA, 236-245.

McCabe, T.J. 1976. A complexity measure. *Software Engineering, IEEE Transactions on* 2: 308-320.

McConnell, S. 2004. High-quality routines In *Code complete 2*. 2nd ed. , 161-186. Redmond, Washington, USA: Microsoft Press.

Mens, T., and Tourwe, T. 2004. A survey of software refactoring. *Software Engineering, IEEE Transactions on* 30: 126-139.

Meulman, J. J. 1998. *Optimal scaling methods for multivariate categorical data analysis*. SPSS, SPSS White Paper (accessed 03/03/2005).

Meulman, J. J., and W. J. Heiser. 2001. *SPSS categories 11.0*. Chicago, USA: SPSS Inc.

Miles, M. B., and M. A. Huberman. 1994. *Qualitative data analysis*. 2nd ed. Thousand Oaks, California, USA: Sage Publications.

Moilanen, T., and S. Roponen. 1994. *Kvalitativiisen aineiston analyysi atlas.ti-ohjelman avulla*. Helsinki, Finland: Kuluttajatutkimuskeskus.

Muthanna, S., Stacey, B., Kontogiannis, K., and Ponnambalam, K. 2000. A maintainability model for industrial software systems using design level metrics. Proceedings of Seventh Working Conference on Reverse Engineering, Brisbane, Australia, 248-256.

Oman, P. W., J. Hagemeister, and D. Ash. 1991. *A definition and taxonomy for software maintainability*. Software Engineering Test Lab, University of Idaho, 91-08.

Oman, P.W., and Hagemeister, J. 1994. Constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software* 24: 251-266.

Pigoski, T. M. 1996. *Practical software maintenance*. John Wiley & Sons Inc.

Rajlich, V.T., and Bennett, K.H. 2000. A staged model for the software life cycle. *Computer* 33: 66-71.

Rombach, D.,H. 1987. Controlled experiment on the impact of software structure on maintainability. *Software Engineering, IEEE Transactions on* 13: 344-354.

Seaman, C.B. 1999. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on* 25: 557-572.

Shepperd, M.J. 1990. System architecture metrics for controlling software maintainability. IEE Colloquium on Software Metrics, , 4/1-4/3.

Shneiderman, B. 1980. *Software psychology: Human factors in computer and information systems.* Cambridge, Massachusetts, USA: Winthrop Publishers.

Siegel, S. 1956. *Nonparametric statistics for the behavioral sciences* 1st ed. New York: McGraw-Hill.

Simon, F., Steinbruckner, F., and Lewerentz, C. 2001. Metrics based refactoring. Proceedings Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, 30-38.

Siy, H., and Votta, L. 2001. Does the modern code inspection have value? , 281-289.

Sommerville, I. 2001. *Software engineering*. Reading, MA, USA: Addison-Wesley / Pearson.

Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., and Russo, B. 2005. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Software Engineering* 10: 81-104.

Szulewski, P.A., and Budlong, F.C. 1996. Metrics for ada 95: Focus on reliability and maintainability. *CrossTalk - the Journal of Defence Software Engineering* 1996.

Tourwé, T., and Mens, T. 2003. Identifying refactoring opportunities using logic meta programming. Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, Benevento, Italy, 91-100.

Vasama, P., and Y. Vartia. 1979. *Johdatus tilastotieteeseen osa 2*. 3rd ed. Pori: Gaedeamus.

Welker, K.D., Oman, P.W., and Atkinson, G.G. 1997. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice* 9: 127-159.

Appendix A – Lecture Slides of Experiment A

This appendix contains the lecture slide of Experiment A.

SoberIT
Software Business and Engineering Institute

Bad Smell in Code – Inspection Exercise

T-76.613 – Software Testing & Quality Assurance
<mika.mantyla@soberit.hut.fi>

SoberIT
Software Business and Engineering Institute

Contents of the lecture

- Bad Code Smells
- Motivation
- Exercise notes & Grading
- Software under study
- Smells under inspection
- Inspection

2

This slide contained the image of shanty town, which are commonly seen in the surroundings of big cities in the countries with lower economical status. For copyright reason the image has been removed.

SoberIT
Software Business and Engineering Institute

Refactoring

- Can be used to avoid structure seen in the previous slide
- Definition of Refactoring adopted from Fowler
  - Controlled way to improve the software's structure without changing its observable behavior
- Benefits of Refactoring according to Fowler and Arnold
  - It improves software design
  - Programs are easier to understand
  - It helps you finding bugs
  - Increased software development speed
  - Easier testing, auditing, documenting
  - Reduced dependency on individuals
  - Greater job satisfaction
  - Extending system's lifetime
  - Preserving software asset value to organization

4

SoberIT
Software Business and Engineering Institute

Bad Code Smells

- Bad Code Smells (Fowler & Beck 2000) are an aid for developers in deciding when software needs refactoring
- Smells are structures that indicate bad software design
  - They were covered in more detail in previous lecture
  - E.g. Large Class, Duplicate Code
  - Of course, the list of bad smells can never be complete
- Why are they called smells?
  - Fowler & Beck think that when it comes to refactoring decision: "*no set of metrics rivals informed human intuition*
- Smells are
  - More solid than common criteria of good programming style
  - Deliberately vague so that human judgment can be applied

5

This contained the image of middle aged nurses smelling at men's armpits likely as a part of some sort of deodorant testing experiment.

SoberIT
Software Business and Engineering Institute

Subjective vs. objective measures of sw structure

- Subjective measures
  - Bad code smells
- Objective measures
  - Automatic measurement with metrics
    - has no wide adaptation
  - Lines of code per method, coupling between objects
- Subjective opinions are more often used in practice
  - Process is often ad hoc or gut feeling based
  - E.g. Programmer constantly evaluates the code under work "This code looks like spaghetti"
    - Especially when working on some else's code
- Very little is known about subjective opinions
  - Conflicting opinions about source code formatting
    - Historically: Curly brace wars have caused some problems
    - Standards and pretty printers have pretty much solved the problem
  - How about software structure?
    - Opinions on: Number of parameters for method, Method size?

7

SoberIT
Software Business and Engineering Institute

Exercise notes

- There are no right answers in this survey
  - We expect that 99% of you will receive full points.
  - However, your opinions should be reasonable
    - For example: Claiming that method with zero parameters has a lot of Long Parameter List smell will not be reasonable
- I will provide group averages after the exercise
  - You can compare your opinions against the others

8

SoberIT
Software Business and Engineering Institute

Grading

- Contents of the exercise
  - inspection and demographic data
- Grading of the course
  - **Maximum 3 points from this exercise**
  - Maximum 40 points come from the exam
  - Maximum 10 points come from weekly exercises

9

SoberIT
Software Business and Engineering Institute

Software under study

- Quick demo
  - This software is only prototype
    - It is lacking huge amount of features
    - In future it should support visualizations of family trees
    - How ever the data model is about complete
- UML-model
  - The visible methods are to be evaluated

10

SoberIT
Software Business and Engineering Institute

Smells under inspection

- All smells under inspection focus at method level
  - There is no time to go through smells involving class relationships

11

SoberIT
Software Business and Engineering Institute

Long Method

- Description:
  - Long methods have low cohesion
    - Low cohesion means that method tries to do several things. Rather than performing just single task
  - Method that is too long is difficult to understand
- Benefits of small methods:
  - Easier to use descriptive method names
    - No need to look at the method body
    - Name tells the purpose of the code
  - Methods that perform single tasks are easier to reuse
- Fix:
  - Split big method into several smaller methods

12

SoberIT
Software Business and Engineering Institute

Long Parameter Lists

- Description:
  - Are constantly changing as more data is needed
  - Are hard to understand
- Before OO everything routine needed was to be passed as parameter
  - The alternative was global data, which usually is even worse
- With OO there really is no need for long parameter lists
  - The method host already has much of the data that method needs
  - You can always ask other objects for more data or/and pass whole objects instead
- Fix:
  - You can always ask other objects for more data or/and pass whole objects instead

13

SoberIT
Software Business and Engineering Institute

Feature Envy

- Description:
  - Method is more interested in other class(es) than the one it is in
  - E.g. If method invokes several getter and setter methods of other classes it is envying data.
  - The key idea of OO is that data and the logic are in the same place
    - Design Patterns don't necessarily follow this principle
    - DP's should be used only when necessary
- Fix:
  - Move method to the class it really should be in

14

SoberIT
Software Business and Engineering Institute

Doing the inspection

- Work alone
- Evaluate each method for all the presented smells
  - Does the smell exist in the method?
    - Scale 1-7 *Not at all - Yes very much*
- Evaluate whether you would refactor this method in order to keep software easy to understand and develop further
- There are total of 10 methods to be evaluated
- For each method you have 5 minutes to evaluate it and answer the questions
  - I will make a note when there is one minute left
  - Do not proceed to the next method before I say you should
  - The time for each method might feel like too long or short

15

SoberIT
Software Business and Engineering Institute

Material

- Everyone should have
  - Smell descriptions (one paper)
  - UML class diagram of the software (one paper)
  - The exercise sheet (11 papers)

16

Appendix B – Web Page of Experiment B

This appendix contains the web-page that was used to give the exercise instruction in Experiment B.

Exercise description

Your job is to familiarize yourself with the software called Family Tree Professional. After getting a general overview of the software you will then review ten methods from the software's source code. During the exercise you will also need to answer some questions concerning your background. After reviewing each method:

* You will indicate whether you would refactor the method or not in order to keep this software easy to understand and develop further.

* You will also need to give a short rationale for you choice. (Short = 1-3 sentences).

* You may answer in Finnish, Swedish or English.

What is refactoring

Definition adapted from [2]: Refactoring is a controlled way to improve the software's structure without changing its observable behavior.

Benefits of refactoring according [1,2]

* Improves software design

* Programs are easier to understand

* Reduced dependency on individuals

* Easier testing, auditing, documenting

* Makes bugs easier to find

* Increased software development speed

* Greater job satisfaction

* Extended system's lifetime

* Preserving software asset value to organization

Thus, by improving the software structure, we should should receive some or all of the above presented benefits.

Grading

The exercise will be graded based on the rationale you provide for your refactoring decision. Strictly speaking there are no right or wrong answers for this exercise.

* You can freely select any of the choices for the refactoring questions as long as you are able to provide good rationale.

* Getting 100% from the exercise should not be very difficult as long as you put some thought and effort writing up your rationale.

* Also try to keep your rationale short. More than 5 sentences is definitely too much

* Finally: we are not interested in what your "guru" buddy thinks. Please answer only based on your own opinions!

Effort

The exercise should take 1-2 hours to complete.

* About 15-30 minutes should be spent in getting an overview of the software by looking at the screen shots, the UML-diagram, and the source code

* About 5-10 minutes to answer the demographic questions.

* Then 40-80 minutes should be spent in evaluating the ten methods. This will mean that you will have 5-10 minutes to study each method and write your rationales. However, the methods are somewhat different so you might spend just few minutes studying the shorter methods.

* Also when you study the methods you will need to get back and forth between the UML and the source code so that you can better understand the method and its context.

How the exercise information may be used

* The refactoring choices for each method will be published through the web so that students can study the opinions of other students. However it will be impossible to recognize individual respondents.

* The rationales for the refactorings will not be published as is. However the issues will be covered during the lecture.

* All the data provided may be used for scientific purposes, but the results will be published in a way that makes it impossible to recognize individual respondents.

Appendix C – Topic Frequencies

This appendix contains the codes discovered from the evaluated software elements, i.e. methods. The software elements are referenced as primary docs 1-10 and they can found in Listings 1-10 in Section 4.3. Codes that have *-mark in front of them indicate super codes that have been created by combining several codes. Combining in this case does not mean adding, e.g. adding number of extractMethod topics and longMethod topics does not give the number of super topic extractMethod+longMethod because many informant specified both topics and such cases are only counted as one.

The topic frequency means the number of informants who mentioned the issue; however, one individual may have specified several topics. Thus, the addition of all topic frequencies can be larger than the number of informants; however number of individual topic mentions can be only as large as the number of informants.

	PRIMARY DOCS										
CODES	1	2	3	4	5	6	7	8	9	10	Totals
*commentsImprove	9	4	1	2	2	1	6	3	7	5	40
*extractMethod + longMethod	0	15	2	16	0	22	0	4	0	18	77
*layoutPoor	1	3	1	4	2	12	2	5	3	4	37
*longStatement + statementSplit	17	0	0	0	2	0	0	12	0	0	31
Totals	27	22	4	22	6	35	8	24	10	27	185

	PRIMARY DOCS										
CODES	1	2	3	4	5	6	7	8	9	10	Totals
callerProblems	1	0	0	0	0	0	1	0	0	0	2
changeAlgorithm	1	8	0	12	3	3	1	3	0	2	33
codingConventions	1	7	1	2	3	0	2	1	2	4	23
coditionalsCombine	0	0	0	0	0	0	0	0	0	2	2
commentsAdd	7	3	1	1	2	1	6	1	6	1	29
commentsOK	0	1	2	0	0	0	1	2	2	0	8
commentsPoor	1	1	0	1	0	0	0	1	1	4	9
commentsPosition	1	1	0	0	0	0	0	1	0	0	3
commentsRemove	2	0	0	0	0	0	0	0	0	0	2
complexLoop	0	0	0	1	0	0	0	0	0	1	2
conditionalsTooMany	0	0	0	0	0	0	0	0	0	2	2
correctness	3	3	1	2	0	0	1	2	2	3	17
coupling	0	0	2	0	0	0	0	0	1	0	3
couplingLow	0	0	0	0	0	0	0	0	1	0	1
duplication	0	0	0	1	0	0	0	3	0	0	4
dynamicConcept	0	0	0	0	0	0	0	0	1	0	1
exctractCommonMethod	0	0	0	0	0	0	0	3	0	0	3
extractClass	0	0	0	0	0	1	0	0	0	0	1
extractMethod	0	13	2	16	0	21	0	4	0	18	74

extractVariable	0	0	0	1	0	1	0	0	0	0	2
genderStoredString	0	0	4	0	0	0	0	0	0	0	4
generalize	0	0	0	0	0	0	0	0	0	1	1
improve	0	0	0	0	0	0	0	0	0	1	1
inlineMethod	0	0	0	0	1	0	0	0	0	0	1
inlineResponsibility	0	0	0	0	0	1	0	0	0	0	1
lacksUnderstanding	0	1	4	0	1	0	1	3	1	0	11
largeClass	0	0	0	0	0	1	0	0	0	0	1
layoutAddBlankLines	0	3	1	4	1	9	0	3	2	3	26
layoutComments	0	1	0	0	0	0	0	0	0	0	1
layoutGroupingGood	0	0	0	0	0	2	0	0	0	0	2
layoutGroupingPoor	0	1	0	1	0	8	0	0	0	1	11
layoutIndent	1	0	0	0	0	0	0	3	0	0	4
layoutMethodArgsMore	0	0	0	0	0	0	2	0	1	0	3
layoutSplitLines	0	1	0	1	0	0	0	0	0	0	2
layoutWhiteSpace	0	1	0	1	1	3	0	0	1	2	9
longMethod	0	9	0	5	0	15	0	1	0	1	31
longStatement	17	0	0	0	2	0	0	10	0	0	29
methodNameOK	0	0	0	0	0	0	1	0	0	0	1
methodNamePoor	0	0	0	0	0	0	0	0	0	2	2
methodVisibility	0	0	0	0	0	0	1	0	0	0	1
moveMethod	2	0	0	0	0	0	0	0	0	0	2
moveResponsiblity	1	0	1	0	0	0	1	0	0	0	3
namingImprovmentOthe	0	0	4	0	0	0	0	0	0	0	4
namingImprovmentVars	0	0	0	3	1	0	0	3	6	2	15
nestingTooMuch	0	0	0	0	0	0	0	0	0	11	11
nonsense	3	1	2	0	1	0	1	1	2	0	11
OK %1	2	1	1	2	5	2	1	4	2	0	20
OK %2	2	5	13	4	16	1	15	3	19	0	78
OK %3	3	1	8	0	6	1	5	0	3	0	27
overloadingNeeded	1	0	0	0	0	0	0	0	0	0	1
parameterListLong	1	0	1	0	0	0	1	0	0	0	3
parameterObject	1	0	1	0	0	0	2	0	0	0	4
parametersAreNotChec	0	0	0	0	0	0	0	0	1	0	1
parametersModify	0	0	1	0	0	0	0	2	0	0	3
performance	0	3	1	0	1	0	0	1	0	0	6
readabilityOK	0	0	2	0	0	1	3	0	1	0	7
readabilityPoor	4	0	1	5	4	18	0	7	1	14	54
refiguringComplete	0	0	0	0	0	0	0	1	0	4	5
reOrganizenternal	0	2	0	2	3	0	2	0	0	1	10
spellingGrammar	0	0	0	1	0	0	6	0	1	0	8
statementSplit	14	0	0	0	1	0	0	8	0	0	23
statmentsMerge	0	1	0	0	0	0	0	0	0	0	1
tempsAdd	0	0	1	1	2	0	0	1	0	0	5
tempsReduce	0	0	0	4	0	0	0	0	0	0	4
Totals	69	68	55	71	54	89	54	72	57	80	669