

Subjective evaluation of software evolvability using code smells: An empirical study

Mika V. Mäntylä · Casper Lassenius

Published online: 27 May 2006

© Springer Science + Business Media, LLC 2006

Editors: Mark Harman, Bogdan Korel, Panos Linos, Audris Mockus and Martin Shepperd

Abstract This paper presents the results of an empirical study on the subjective evaluation of code smells that identify poorly evolvable structures in software. We propose use of the term software evolvability to describe the ease of further developing a piece of software and outline the research area based on four different viewpoints. Furthermore, we describe the differences between human evaluations and automatic program analysis based on software evolvability metrics. The empirical component is based on a case study in a Finnish software product company, in which we studied two topics. First, we looked at the effect of the evaluator when subjectively evaluating the existence of smells in code modules. We found that the use of smells for code evaluation purposes can be difficult due to conflicting perceptions of different evaluators. However, the demographics of the evaluators partly explain the variation. Second, we applied selected source code metrics for identifying four smells and compared these results to the subjective evaluations. The metrics based on automatic program analysis and the human-based smell evaluations did not fully correlate. Based upon our results, we suggest that organizations should make decisions regarding software evolvability improvement based on a combination of subjective evaluations and code metrics. Due to the limitations of the study we also recognize the need for conducting more refined studies and experiments in the area of software evolvability.

Keywords Code smells · Subjective evaluation · Perceived evaluation · Maintainability · Evolvability · Code metrics · Software metrics · Human factors

1. Introduction

Many software-related tools, processes, methodologies and techniques claim to improve or assess software quality. Yet, software quality, as quality in general, is

M. V. Mäntylä (✉) · C. Lassenius
Helsinki University of Technology, Helsinki, Finland
e-mail: mika.mantyla@tkk.fi

C. Lassenius
e-mail: casper.lassenius@tkk.fi

largely dependent on one's point of view (Garvin 1984; Kitchenham and Pfleeger, 1996). From a developer's viewpoint the ease of modifying and further developing the software is clearly one of the most important quality dimensions. Traditionally one would call this viewpoint to quality software maintainability.

In order to be able to improve software maintainability, its current state and possible improvement efforts must be made measurable. However, assessing maintainability is a difficult task. Widely studied object-oriented metrics (Briand et al., 1997, 1999; Chidamber and Kemerer, 1994; Harrison et al., 1998; Henderson-Sellers, 1996; Hitz and Montazeri, 1996; Lorenz and Kidd, 1994) offer a way to assess software maintainability. Although some studies report successes in measuring maintainability with metrics (Bansiya and David, 2002; Chidamber et al., 1998; Coleman et al., 1994; Li and Henry 1993; Subramanyam and Krishnan, 2003), there are people, especially in the agile community (Beck et al., 2001), who are critical to using metrics for maintainability assessment, since they feel that metrics fail to account for the breadth of issues that humans consider when evaluating code.

We have noticed—by studying several small and medium-sized Finnish software product companies—that design and source code metrics are not widely utilized in practice. Despite this, in our experience, also small companies suffer from harmful effects of software evolution (Lehman, 1980) that cause system quality degeneration. Fighting against the effects of software evolution is especially important in the software product business, where products evolve in sequential releases. Cusumano and Selby (Cusumano and Selby, 1995) describe how Microsoft uses 20% of its development effort to re-develop the code base of its products. Cusumano and Yoffie (Cusumano and Yoffie, 1998) report how Netscape's inability to refactor the code base hindered their software development, and how Microsoft's redesign efforts in the Internet Explorer 3.0 project later paid off.

The agile community, which emphasizes individuals and interactions over tools and processes, has come up with a term called *code smell* (Fowler and Beck, 2000) to help software developers recognise problematic code. These code smells are general descriptions of bad code that are supposed to help software developers decide when the code needs refactoring. In most cases, the goal of code refactoring is to make the software easier to understand and/or extend. Fowler and Beck (Fowler and Beck, 2000) claim that exact criteria cannot be given to determine when a software needs refactoring. In their words, when it comes to making refactoring decisions: “no set of metrics rivals informed human intuition.” This opinion is in conflict with the idea that the level of software maintainability can be determined with source code or design metrics. A counterexample to Fowler and Beck is provided by Grady (Grady, 1994), who reports that in some HP divisions there were tight threshold limits that a program was not allowed to exceed, e.g., a Fortran program's cyclomatic complexity (McCabe, 1976) should not exceed 15. This threshold was determined based on data from previous software projects. Also, the idea of measuring software maintainability with metrics (Coleman et al., 1994, 1995) seems to be conflicting with the idea of assessing maintainability using vaguely defined qualitative criteria, such as the code smells of Fowler and Beck.

To our knowledge, others have not published studies in which the code smells suggested by the agile community would have been used as a basis for subjective code evaluation. Development-level anti-patterns (Brown et al., 1998), which have some overlap with code smells, also seem to be lacking research.

This paper describes our initial efforts in empirically studying the use of the code smells. In this paper we look at the uniformity of smell evaluations, the effect of the evaluators' demographics on the evaluations, and compare the smell evaluations against selected source code metrics. Despite the very tentative nature of our findings, we hope that they help stimulate more empirical research aiming at critically evaluating, validating and improving our understanding of subjective indicators of software maintainability.

The rest of the paper is structured in the following way: Section 2 provides the theoretical background. Section 3 describes the research methods and objectives. In Sections 4 and 5 we present the results of the study. Section 6 provides the discussion of the results and addresses the limitations of the study. Finally, Section 7 presents the conclusions of this study and suggestions for future work.

2. Theoretical Background

This section provides the background to our work. First, we argue in favour of using the term software evolvability instead of the term software maintenance. Section 2.2 introduces four viewpoints to software evolvability, and Section 2.3 focuses on one of these viewpoints, i.e., *software evolvability evaluation*. Finally, Section 2.4 focuses on the prior work on human-based software evolvability evaluation.

2.1. Software Evolvability

In this paper 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 dictionary¹ 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

¹ <http://www.m-w.com/>

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 (pp. 660, 2001). 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.

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. (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

² Unfortunately we have not been able to obtain a copy of the report. A brief summary can be found in (Pigoski, 1996) on page 288.

Table 1 Viewpoints to software evolvability

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

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 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 address. The focus of this paper is on the *evaluation of software evolvability*.

2.3. Approaches to Software Evolvability Evaluation

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, 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 it is the developer who makes the decision 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 Fig. 1. The figure suggests that program analysis offers a quantitative and objective way to analyze software quality. Human evaluations on the other hand are always more or less subjective, but they offer qualitative information about the software which cannot easily be obtained by program analysis tools. Humans can also consider aspects that are not included in the predefined metrics calculated by tools.

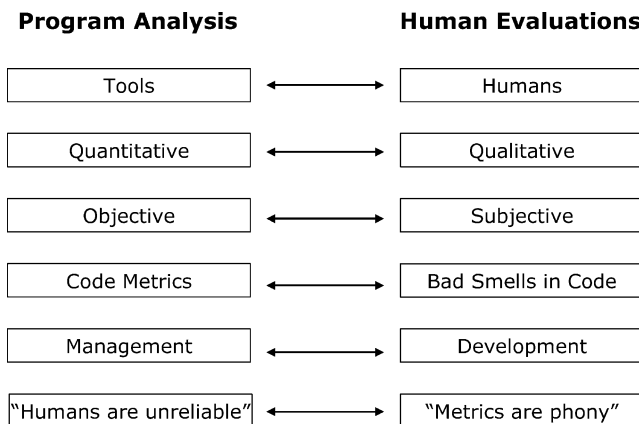


Fig. 1 Differences between automatic program analysis and human evaluations

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.

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, e.g., (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 using historical data to detect spots where refactorings have 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 improvement of evolvability than on its evaluation. We have been able to find only a limited number of studies in which subjective evolvability evaluations have been studied or compared to automatic program analysis. In the following subsections we introduce the relevant prior work discussing subjective evolvability evaluation.

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 (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 somewhat 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” by 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 (GodClasses), 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 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 (1980, pp. 134–138) reports 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 Likert 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 also 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 aerospace 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.

Oman et al. (Coleman et al., 1994, 1995; Oman and Hagemester, 1994; Welker et al., 1997) report on the construction of a maintainability index. In this 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 1,000 to 10,000 lines of code (Oman and Hagemester, 1994). After calibration, they performed a validation study where they again acquired opinions and the source code on six industrial systems ranging from 1,000 to 8,000 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 Hagemester, 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 effectively 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).

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.

Four out of the five referred studies are not made with object-oriented languages, which currently dominate the field of software development. The drawback in the only study involving object-oriented software (Kataoka et al., 2002) is that the data set consists of only five refactorings and that only one developer evaluated the effectiveness of the refactorings. Although there are some studies on subjective design/code quality evaluation and the evaluations have been compared with source code metrics, we feel that there is still ample research space to be filled.

3. Research Methods and Objectives

This section introduces the research methods and objectives. Section 3.2 discusses the research objectives and research questions. Section 3.2 focuses on the research methodology and data analysis techniques used. Sections 3.3 and 3.4 introduce the case company and the informants. Finally, Section 3.5 presents the smells that were studied.

3.1. Research Objectives

Our research objective was to increase our understanding on subjective software evolvability evaluation. Our study consisted of two parts. First, we were interested in the possible variations in the subjective evaluations and the explaining factors behind these variations. Second, we wanted to understand the relationship between subjective software evolvability evaluations and source code metrics. Altogether, we had three research questions:

1. Do software developers have a uniform opinion on the “smelliness” of the source code?
2. Do the demographics of the developers affect the smell evaluations, and if so, how?
3. Do the developers’ evaluations on code smells correlate with related source code metrics?

The motivation for research questions one and two comes from Fowler and Beck’s (2000) idea that no precise criteria for evaluating code smells can be given. Since human judgement plays a significant role when evaluating smells, it is interesting to see if the evaluations are uniform. The developers should have a common view, otherwise the usability of code smells as indicators of software evolvability is questionable. Research question 2 addresses the concern that the demographic variables, such as experience, affect the smell evaluations given by a developer.

By answering the third research question, we aim at understanding whether the smell evaluations on different modules correlate with the source code metrics for a particular smell. This is essential, because if the human evaluations and source code metrics provided by the program analysis do not correlate, it indicates that one of them is failing to provide the correct results.

3.2. Research Methodology and Data Analysis

We answer the research questions based on the data provided by a survey in which we asked industrial software developers to evaluate how much of each code smell existed in a particular software module of a software product they are familiar with. All informants were software developers in a Finnish software company, which we will call BeachPark. Each informant answered the questions only regarding the software modules he or she was familiar with. In addition to the survey, we also gathered data through informal discussions with the case company.

We collected the data using a two-page web-based questionnaire. On the first page, each respondent provided background information, including age, location, role (developer or lead developer), education, work experience in the company, and overall software development work experience. The respondents also indicated the software modules they had primarily worked with.

In the second page of the survey, the respondents assessed the degree of various code smells for the modules they selected as the modules they had primarily worked with. We asked about 23 code smells, which were described with a definition and an example, totalling about 35 words each. The assessments were made on a seven-point Likert scale, with 1 indicating a total lack of the smell, and 7 indicating a major presence of the smell in the evaluated code. The assessment was made for each module-smell pair. The respondents could also select “I don’t know” or “I don’t understand the smell description.” The “I don’t know” option was checked by default to prevent wrong smell evaluations. The respondents also estimated how well they knew each module they had selected. The scale was also from 1 to 7 where 1 meant “I know the module very poorly” and 7 meant “I know the module very well.”

We used the SPSS™ program intended for statistical analysis to analyze the answers. In addition to traditional values such as mean and standard deviation, we also used the Mann–Whitney test to study differences in evaluations between the various demographic groups, for example developers and lead-developers. This test is characterized by Siegel (Siegel, 1956) as follows: “When at least ordinal measurement has been achieved, the Mann–Whitney U test may be used to test whether two independent groups have been drawn from the same population. This is one of the most powerful of the nonparametric tests, and it is a most useful alternative to the parametric t test when the researcher wishes to avoid the t test’s assumptions, or when the measurement in the research is weaker than interval scaling.”

Furthermore, some of the data analysis decisions are explained in the Results section. For example, the decisions why some analyses are performed only on certain software modules and the discussion on the various source code metric thresholds for the smells are covered in more detail in that section. We have made this decision in order to enhance the readability of the paper, because separating,

e.g., the metric threshold discussion from the use of metrics would require the reader to browse back and forth.

3.3. Case Company

BeachPark is a small Finnish software product development company developing non-domain-specific products. At the time of the survey, the company employed 18 software developers. The company had developed two software products in the past 4–5 years and during that time some parts of the products had become complex and thus were difficult to work with. The development language used in the products was Delphi, which is an object-oriented extension of the Pascal programming language. The software modules and their sizes are listed in Table 2 in lines of code (LOC) which throughout this work means non-blank, non-commented lines of source code. The module names beginning with an “A” or “B” represent product-specific modules and the module names beginning with a “Y” are shared between the two products. The module sizes marked with an asterisk are estimates made by the case company employees. However, when comparing the employee size estimates to the actual size of the modules that we had access to, we noticed that the employee size estimates tended to be a bit too high. Thus, the size estimates must be considered only as indicators of the module size.

The research questions listed in Section 3.1 are answered based on the data gathered from the modules listed in Table 2. Section 4.1 provides the results for research question 1, based on modules A1, A2, B1, B3, Y1, Y2, and Y4. Section 4.2 provides the results for the second research question based on all the modules listed in Table 2. Finally, the results for the third research question are based on only three modules, A1, A2, and A3, since we did not have access to the source code for the other modules.

3.4. Informants

Table 3 shows the age, role, education, work experience, and modules evaluated of our 12 informants. In the table we can see that in general the developers of the

Table 2 The software modules

Name	Size in LOC	Age (years)	Number of evaluators
A1	83 200	6–7	5
A2	28 654	2–3	6
A3	16 787	<1	1
B1	55 342	5	3
B2	Unknown	2–3	4
B3	54 780	5	4
Y1	80 000*	1–2	3
Y2	30 000*	1–2	3
Y3	20 000*	1–2	4
Y4	20 000*	Unknown	3
Y5**	Unknown	Unknown	1

*Employee estimates.

**Currently obsolete.

Table 3 Demographic data about the respondents (all numbers are in years)

Id	Age	Role	Education	Development experience in company	Development experience total	Modules evaluated (knowledge of each module 1 = lowest and 7 = highest)
1	23	Developer	Student M.Sc.	2.50	2.50	B3(5), B1(6), B2(3)
2	26	Developer	B.Sc.	3.42	3.42	B3(6), B1(7)
3	37	Lead developer	Student M.Sc.	7.00	7.00	Y5(6), Y1(5), Y2(7), Y3(7), Y4(6)
4	25	Developer	Student M.Sc.	2.58	2.58	A2(4), B2(3) A1(5)
5	27	Lead developer	M.Sc.	2.50	6.42	Y1(7), A2(4), Y3(5), Y4(6)
6	32	Lead developer	M.Sc.	6.92	8.00	A3(7), A2(6), B3(5), Y3(6), Y4(5), A1(7)
7	25	Developer	Student M.Sc.	3.50	3.50	Y1(6), B3(5), B2(7)
8	25	Developer	Student M.Sc.	3.00	3.00	A2(6), A1(5)
9	27	Developer	Student M.Sc.	1.42	3.00	Y2(6), Y3(6)
10	31	Lead developer	Student M.Sc.	4.83	4.83	A2(6), A1(7)
11	26	Developer	Student B.Sc.	1.83	2.67	B1(6), B2(4), Y2(3)
12	40	Developer	B.Sc.	1.50	10.00	A2(5), A1(5)

company were quite young with a median age of 26.5 years. In this survey, there were no informants that had recently joined the company. This is positive, because answering the questionnaire required considerable knowledge of the software under investigation. All informants were professionals, with several years of professional programming experience. The education of the informants was distributed as follows: two had a master's degree, seven were currently studying for a master's degree, two had a bachelor's degree, and one was studying for a bachelor's degree. There were four lead developers and eight regular developers. The division by geographical location (the organization had software developers at two physical locations) was somewhat unbalanced, as only three answers came from one location while the other location generated nine answers.

We achieved a response rate of 67%—12 out of the 18 developers working for the company participated in the survey. All in all we received 37 module-smell evaluations from the 12 developers. Thus, the average number of modules evaluated by each developer was about three, varying from 2 to 6. The number of evaluations per module fluctuated from 1 to 6 and the average was 3.4.

3.5. Introducing the Smells under Study

In this section we introduce the 23 code smells that we surveyed. Twenty-two of the code smells were introduced by Fowler and Beck (2000). Additionally, we included a code smell for dead code, i.e., code that is never executed, as we felt it was important but not included in the list by Fowler and Beck. In the rest of the paper we will use a bold font when referring to the smells, e.g., *Dead Code*. We chose four smells for automatic code analysis, namely *Large Class*, *Long Method*, *Long Parameter List*, and *Duplicate Code*. We selected these because we thought their operationalization would be quite straightforward, and because we had suitable tools to measure them. A list of 23 code smells can be difficult to understand. Hence,

we grouped the smells into six categories. The taxonomy is an improved version of our previous taxonomy that was presented in (Mäntylä et al., 2003). An alternative grouping for the code smells can be found for example in (Wake, 2003).

The Bloater smells are: *Long Method*, *Large Class*, *Primitive Obsession*, *Long Parameter List*, and *Data Clumps*. Bloater smells represent something that has grown so large that it cannot be effectively handled. It seems likely that these smells grow a little bit at a time. Hopefully nobody designs, e.g., lengthy methods.³ *Primitive Obsession* is actually more of a symptom that causes bloats than a bloat itself. The same holds for *Data Clumps*. When a *Primitive Obsession* exists, there are no small classes for small entities (e.g., phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software. With *Data Clumps* there exists a set of primitives that always appear together (e.g., three integers for RGB colours). Since these data items are not encapsulated in a class this increases the sizes of methods and classes.

The Object-Orientation Abusers are: *Switch Statements*, *Temporary Field*, *Refused Bequest*, and *Alternative Classes with Different Interfaces*. The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design. For example, using a switch statement to detect type code, i.e., simulate a class hierarchy, might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. The *Refused Bequest* smell lacks proper inheritance design, which is one of the key elements in object-oriented programming. The *Alternative Classes with Different Interfaces* smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The *Temporary Field* smell means a case in which a variable is in the class scope, when it should be in the method scope. This violates the information hiding principle.

The Change Preventers are: *Divergent Change*, *Shotgun Surgery*, and *Parallel Inheritance Hierarchies*. Change Preventers are smells that hinder changing or further developing the software. These smells violate the rule suggested by Fowler and Beck, which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, and changes to calculation formulas only affect one other class. The *Divergent Change* smell means that we have a single class that needs to be modified by many different types of changes. With the *Shotgun Surgery* smell, the situation is the opposite—we need to modify many classes when making a single change to a system, e.g., change several classes when changing the database from one vendor's to another's. *Parallel Inheritance Hierarchies*, which means a duplicated class hierarchy, was originally placed in the OO-abusers. One could also place it inside the Dispensables, since there is redundant logic that should be replaced. However, we have placed it here because in *Parallel Inheritance Hierarchies*, classes and changes do not have one-to-one relationship, e.g., with parallel hierarchies there is always more than one class that requires changing.

³ A recent study (Robillard et al., 2004) of program modification tasks also showed that programmers who performed their modifications inside few methods were less successful than the ones who distributed their solutions to several methods.

The Dispensables are: *Lazy class*, *Data class*, *Duplicate Code*, *Dead Code*, and *Speculative Generality*. The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code. This group contains two types of smells, dispensable classes and dispensable code, but since they violate the same principle, we look at them together. If a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with the *Lazy class* and the *Data class* smells. Code that is not used or is redundant needs to be removed. This is the case with the *Duplicate Code*, *Speculative Generality* and *Dead Code* smells.

The Couplers are: *Feature Envy*, *Inappropriate Intimacy*, *Message Chains*, and *Middle Man*: This group has four coupling-related smells. One design principle that has been around for decades is low coupling (Stevens et al., 1974). This group has three smells that represent high coupling. The *Middle Man* smell on the other hand represents a problem that might be created when trying to avoid high coupling with constant delegation. A *Middle Man* is a class that is doing too much simple delegation instead of really contributing to the application. The *Feature Envy* smell means a case where one method is heavily coupled to other classes than the one that it is in. The *Inappropriate Intimacy* smell means that two classes are coupled tightly to each other. *Message Chains* is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B, while A and B have a direct reference. When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. The following example illustrates the *Message Chains* smell: `A.getB().getC().getD().getTheNeededData()`. Of course, we could make an argument that these smells should belong to the Object-Oriented abusers group, but since they all focus strictly on coupling, we think it makes the taxonomy more understandable if they are introduced in a group of their own.

Other smells that were studied, but could not be included in any of the groups are *Comments* and *Incomplete Library Class*. The *Comments* smell indicates the misuse of comments, i.e., when a programmer uses comments to explain some piece of code instead of writing code that is simpler and more self-explaining. *Incomplete Library Class* means that the application is using a third party library code that is not completely adequate for the application. According to Fowler and Beck this smell can be removed, for example, by introducing a local extension.

4. Results—Effect of Human Factors in Code Smell Evaluation

One key assumption of code smells is that there are no exact conditions, which indicate when developers should improve the software design. Instead, individuals should be able to make the decision primarily using their own intuition. As software developers may have different predilections it is not evident that all developers see the code smells equally.

4.1. Uniformity of the Smell Evaluations

In this section, we describe how uniform the smell evaluations were for the different software modules. Naturally the modules that received the highest number of evaluations are of the greatest interest. In Table 4 we can see the smell evaluations

Table 4 Smell evaluations of modules A1 and A2, scale 1–7 (a larger number indicates more of the smell)

Smell name	Number of answers on a scale of 1 to 7						
	1	2	3	4	5	6	7
Module A1							
Long Method						5	
Large Class				1	3		1
Long Parameter List		1	2	1	1		
Data Clumps			2	1			
Duplicate Code		1	2		2		
Dead Code		1	2	1	1		
Speculative Generality		1		3			
Feature Envy		1	2	1	1		
Inappropriate Intimacy		2		1	1		1
Message Chains		1			3	1	
Middle Man		2	2	1			
Lazy Class	2	1	2				
Data Class	2	1	1				
Incomplete Library Class			2			1	
Primitive Obsession	1	2		1	1		
Switch Statement	1			1	2	1	
Temp Field		2	2		1		
Refused Bequest	1	3	1				
Alternative Classes with Different Interface	1	1	2	1			
Parallel Inheritance Hierarchies	2	1	1	1			
Divergent Change	1	1		2	1		
Shotgun Surgery		2	1	1	1		
Comments		1	1	2	1		
Module A2							
Long Method				2	1	3	
Large Class				2	2	1	1
Long Parameter List		1	1	1	3		
Data Clumps			2		2		
Duplicate Code			3	1	1	1	
Dead Code		4	1			1	
Speculative Generality		2	1	1	1		
Feature Envy		2	1	1	2		
Inappropriate Intimacy		2	1	1	1		1
Message Chains				3	2		1
Middle Man		1	3	1		1	
Lazy Class	2	3	1				
Data Class	2	1	1		1		
Incomplete Library Class			3	2		1	
Primitive Obsession	1	3		1	1		
Switch Statement	1	1	1		2	1	
Temp Field		4	1	1			
Refused Bequest	1	4	1				
Alternative Classes with Different Interface	1	2	2	1			
Parallel Inheritance Hierarchies	2	1	2	1			
Divergent Change	1	1		1	3		
Shotgun Surgery		2	1	2	1		
Comments		2	1	1	2		

of modules A1 and A2 that had the largest number of evaluations. In the *Long Method* smell evaluations for module A1 there is a perfect agreement in the evaluations. However, this case is unique, since the other smell evaluations had larger distributions. For example, there is a large distribution regarding the existence of the *Inappropriate Intimacy* smell in both modules A1 and A2.

To summarise the data seen in Table 4, and to be able to compare it with other modules, we have created Table 6, where we show the standard deviations of the smell evaluations for modules A1, A2, B1, B3, Y1, Y3, and Y4. All these modules had three or more evaluators and they are therefore selected for analysis. Modules B2 and Y2 also had more than three evaluators, but they did not have three or more evaluators that had specified their knowledge of the system as higher than four on the seven-point Likert scale. This lack of knowledge of the modules also resulted in many unanswered questions by the evaluators. Thus, we discarded modules B2 and Y2 from this analysis. A summary of the knowledge levels of the evaluators is show in Table 5. In Table 6, we have marked the smell evaluations with less than three evaluators with a dash (“-”).

In Table 6 we can see that nearly 75% (17 out of 23) of the means distributions are between 0.85 and 1.25. The smells *Switch Statement*, *Inappropriate Intimacy*, and *Message Chains* have a higher distribution than the rest of the smells. Based on the data we cannot offer a good explanation as to why these smells have a higher distribution. The smells that have a very low distribution, less than 0.85, are *Refused Bequest*, *Parallel Inheritance Hierarchies*, and *Comments*. The reason for the low distribution was in some cases evident from the data. We discovered that the evaluations on the smells *Refused Bequest*, *Parallel Inheritance Hierarchies* were all very low, which caused the low distribution. For *Comments* on the other hand most of the evaluations, nearly 70% in all the data, were either two or three on the ordinal whereas only 5% of the respondents had given a one on the ordinal scale.

In the next section we try to explain the deviations in the smell evaluations by studying how the informants made their evaluations and how their demographics affected their evaluations.

4.2. Effect of Demographics

In this section we look at how demographic variables such as module knowledge, role, and work experience affected the informants’ smell evaluations. To study the

Table 5 The knowledge level of the informants (a larger number indicates better knowledge)

Modules	Number of answers per knowledge level						
	1	2	3	4	5	6	7
A1					3		2
A2				2	1	3	
B1						2	1
B2			2	1			1
B3					3	1	
Y1					1	1	1
Y2			1			1	1
Y3					1	2	1
Y4					1	2	

Table 6 The standard deviation of evaluations in different software modules

	A1	A2	B1	B3	Y1	Y3	Y4	Mean
Long Method	0.000	0.983	2.309	1.708	0.577	0.816	1.000	1.056
Large Class	1.095	1.169	0.577	1.258	0.577	0.957	1.528	1.023
Long Parameter List	1.140	1.265	0.000	2.062	0.577	1.000	1.155	1.028
Data Clumps	0.577	1.155	0.577	1.528	–	0.577	0.707	0.854
Duplicate Code	1.342	1.265	1.528	1.291	1.155	0.500	1.155	1.176
Dead Code	1.140	1.602	0.577	1.708	0.577	1.000	1.155	1.108
Speculative Generality	1.000	1.304	0.577	0.500	1.528	0.500	0.577	0.855
Feature Envy	1.140	1.378	0.577	0.500	–	1.000	0.577	0.862
Inappropriate Intimacy	2.121	1.941	–	1.500	0.577	1.258	1.000	1.400
Message Chains	1.095	1.169	1.732	1.708	1.000	1.708	0.577	1.284
Middle Man	0.837	1.378	1.000	0.577	1.000	0.577	1.155	0.932
Lazy Class	1.000	0.753	0.000	0.500	1.528	0.500	2.309	0.941
Data Class	0.957	1.673	1.000	1.155	0.577	1.258	2.000	1.231
Incomplete Library Class	1.225	1.169	0.577	0.577	1.155	0.577	1.155	0.919
Primitive Obsession	1.643	1.506	0.577	0.957	–	0.577	0.577	0.973
Switch Statement	1.924	1.966	3.055	1.732	–	2.082	0.577	1.889
Temp Field	1.225	0.837	2.646	0.000	–	2.708	0.000	1.236
Refused Bequest	0.707	0.632	0.577	0.577	–	1.528	0.707	0.788
Alternative Classes with Different Interface	1.140	1.049	–	0.577	–	–	1.155	0.980
Parallel Inheritance Hierarchies	1.304	1.211	–	0.500	–	0.577	0.577	0.834
Divergent Change	1.643	1.751	–	0.500	1.155	1.000	0.577	1.104
Shotgun Surgery	1.304	1.211	0.577	1.291	1.155	1.000	0.577	1.016
Comments	1.140	1.378	0.577	0.816	–	1.000	0.000	0.819

differences in these background variables, we used the nonparametric Mann–Whitney U test. Tables 7, 8 and 9 show the results of the Mann–Whitney U test, the significance of U , and the number of evaluations of the compared groups.

First, we look at how the informants' role in the organization affected the smell evaluations. The informants consisted of eight developers and four lead developers. However, the lead developers had evaluated a larger number of modules, which meant that we got roughly an equal number of evaluations from regular developers and lead developers. Table 7 shows the smells for which there were significant ($p < 0.05$) differences between these two groups. The regular developers reported higher degrees of *Duplicate Code* than the lead developers. The lead developers on the other hand reported more *Parallel Inheritance Hierarchies* than the regular developers ($p < 0.01$).

The knowledge of a module was measured subjectively by self assessment on a seven-point Likert scale. To study how the knowledge of a module affected the

Table 7 Mann–Whitney U test results between developers and lead developers

	Duplicate code	Parallel inheritance hierarchies
Number of evaluations of <i>regular</i> developers ($n = 8$)	20	17
Number of evaluations of <i>lead</i> developers ($n = 4$)	17	16
Mann–Whitney U	104.000	69.500
p -value (asym. sig., 2-tailed)	0.037	0.009

Table 8 Mann–Whitney U test results between high and low knowledge groups

	Lazy class
Number of evaluations of <i>low</i> knowledge informants	16
Number of evaluations of <i>high</i> knowledge informants	20
Mann–Whitney U	84.000
p -value (asym. sig., 2-tailed)	0.008

smell evaluations, we created two knowledge groups, one with a knowledge value between 3 and 5 (low knowledge) and the other with knowledge values 6 and 7 (high knowledge). There were no answers where the developer’s knowledge of a module was under 3.

Table 8 shows the only smell, *Lazy Class*, for which the evaluations between the high and low knowledge groups were significantly ($p < 0.05$) different. The high knowledge group reported more *Lazy Class* smells than the low knowledge group. *Lazy Class* is a class that is so small that it “does not carry its own weight” and therefore should be removed.

We also studied the effect of work experience on the smell evaluations. We tried different groupings for the work experience variable. The one that provided the most interesting results compared the two informants with the longest work experience in the case company against the rest of the informants. These two informants had worked in the company for seven years, whereas the most experienced of the other informants had been with the company for a little less than five years and the rest of the informants three and a half years or less. The two most experienced informants were the only developers who had been working in the company for the entire lifetime of the software products. The two most experienced informants had also provided 11 smell evaluations, which made the comparison to the rest of the informants sensible.

In Table 9 we can see the smells and the smell index (mean from all the smell evaluations for a module by an informant) that had a significant ($p < 0.05$) difference between the two most experienced developers and the rest. All the significant differences in Table 9 mean that the two most experienced informants in the company evaluated that there is considerably less smells compared to the evaluations of the rest of the informants.

Finally, we studied the differences between the developers’ attitudes regarding the degree of smells. We were able to compare five informants that all had evaluated two particular modules, namely A1 and A2, and noticed that one developer clearly had the most positive attitude towards the quality of the code. His

Table 9 Mann–Whitney U test results between the most experienced developers and the rest

	Smell index	Large class	Feature envy	Data clumps	Alternative class with different interface	Long parameter list
Number of evaluations of <i>most</i> experienced developers ($n = 2$)	11	10	10	8	9	11
Number of evaluations of <i>less</i> experienced developers ($n = 10$)	26	26	25	18	21	26
Mann–Whitney U	65.500	57.000	63.5000	12.000	43.000	85.000
p -value (asym. sig., 2-tailed)	0.010	0.008	0.020	0.002	0.015	0.039

smell index for module A1 was 2.68 and for A2 2.64. This developer gave considerably lower smell evaluations for the two modules than the other four. On the other hand, another developer saw considerably more smells in the two modules than the rest. His smell index for the modules was 4.14 for A1 and 4.27 for A2. The three informants that were in the middle with their smell evaluations had smell indexes between 3.48 and 3.76 for A1 and between 3.13 and 3.76 for A2. Therefore, it seems that the different attitudes of the developers can partly explain the differences in their smell evaluations.

5. Results—Human Evaluation and Source Code Metrics

In this section, we compare the subjective smell evaluations with the source code metrics. For each smell, we first discuss its appropriate metrics, decide the thresholds, and then compare the measurement results to the evaluations. The source code/design metrics for identifying each smell are selected based on the literature as well as on our own understanding. The comparison is limited to only four smells and three modules. The smells are *Large Class*, *Long Method*, *Long Parameter List* and *Duplicate Code*. These were selected because they are quite easy to measure with tools. The modules are A1, A2, and A3, which were the only ones whose source code we were allowed to analyze.

We used an enhanced version of a tool called *same*⁴ to measure the number of duplicate code lines. Technically, *same* does not measure duplicate code lines, since it only investigates whether the lines are lexically identical after removal of whitespaces. This means that *same* will not recognize all possible duplicated lines, e.g., if the variables have different names, and thus gives somewhat conservative results. To gather measures for the *Large Class* and *Long Parameter List* smells, we used a tool called *SDMetrics*.⁵ Additional measures for the *Large Class* smell and all measures for the *Long Method* smell were collected with a tool called Pascal Analyzer version 3 by Peganza Corporation.

5.1. Large Class

The first question arising when working with the *Large Class* smell is: “What exactly is a large class?” Fowler and Beck (2000) say that a large class can often be spotted by looking at the number of instance variables. Therefore, the *number of attributes* is used as one of our class size measures. The *Large Class* smell is also recognized as an anti-pattern known as the Blob, Winnebago, and the God Class. The book that describes this anti-pattern in detail (Brown et al., 1998, pp. 73–84) points out that the *number of methods* and variables provide good measures for such anti-patterns. The number of methods measure was also used in an object-oriented design quality assessment by Bansiya and David (2002). Chidamber and Kemerer (1994) introduce the sum of method complexities in a class as a metric called weighted methods per class (WMC). Chidamber and Kemerer do not state how the complexity of a

⁴ <http://sourceforge.net/projects/same>

⁵ <http://www.sdmetrics.com/>

method should be calculated, but purposefully leave it as an implementation decision. In this work, we actually have measured three versions of the WMC metric, although they are not presented as such in the measurement below (Table 10). Firstly, each method is given a complexity of one, which results in the already discussed methods per class metric. Secondly, we have chosen to use two widely used metrics, lines of code (LOC) and cyclomatic complexity (CC) (McCabe, 1976) to represent method complexity. These two metrics result in a *sum of lines of code of methods per class* and *sum of cyclomatic complexity of methods per class*, respectively. Finally, we feel that class cohesion is a good measure for the *Large Class* smell because large classes often try to do too many things, which indicates low cohesion. Therefore, we have selected to measure *lack of cohesion methods* (LCOM), which was introduced by Chidamber and Kemerer.

We also needed some thresholds for the *Large Class* smell. The anti-pattern book (Brown et al., 1998) refers to “AntiPattern Session Notes” held by Michael Akroyd, who, according to Brown et al. said that a class with more than 60 variables and methods often indicates the presence of the Blob. Much tighter thresholds are presented by Lorenz and Kidd (1994), who suggest a threshold of 3 for instance variables in a model class, and 9 for a user interface (UI) class. They also suggest that a model class should not have more than 20 methods and a UI class should have a maximum of 40 methods. Most academic studies do not provide any thresholds for class size. Based on the suggestions and the measurement we made, we selected three limits for a number of *Large Class* variables and methods. For the number of variables, we used limits of 10, 20, and 40 variables, and for the number of methods, the limits were 30, 50, and 100. For the sum of the lines of code of methods per class we selected three thresholds, 500, 1,000, and 2,000. According to the Java coding standard (Sun Microsystems, 1999), files (Java has typically only one class per file) with more than 2,000 lines are cumbersome. However, this coding standard is not talking about lines of code, but lines of file in general, which includes comments, code outside methods, e.g., import/include/uses statements, and introduction of class variables. Therefore, we selected a lower threshold of 1,000 lines of code. Based on our measurement with these two thresholds we decided that a lower threshold of 500 lines of code was needed. We are not aware of any recommendations for the sum of cyclomatic complexity of methods per class, but there are thresholds for cyclomatic complexity per routine in structural programming. McCabe suggested that with procedural programs the threshold of a program cyclomatic complexity should be 10. In our data, the median number of methods per class in module A1 was 13, in A2 it was 9, and in A3 it was 6. Because A1 had more classes than A2 and A3, the overall median of the number of methods per class was 11. Thus, a typical class in our data had roughly ten methods. We think that a class can be considered too big if it has a typical number of methods (10) that all have a cyclomatic complexity of ten. This results in a threshold of 100 for the sum of cyclomatic complexity of methods per class. As our measurement showed that there were a considerable amount of classes above the threshold of 100, we also selected larger thresholds of 200 and 400 for the sum of cyclomatic complexity. The LCOM measure only has values above zero when a class is not highly cohesive. Therefore, our first threshold is values greater than zero, which in practice means one. As there were many classes above the threshold of one, we also needed to set higher thresholds, but unfortunately for LCOM there exists no threshold above one that we are aware of. Therefore we set the two thresholds to

percentiles of 80 and 90 for all the classes in modules A1, A2, and A3. This resulted in thresholds of 10 and 15, respectively.

The metrics related to the class size of the different modules are collected in Table 10 under the metric data section. The data shows that module A1 clearly has the largest classes, measured by the number of variables. When measuring class size with the number of methods, we can see that modules A1 and A3 have roughly the same amount of large classes. Overall, it looks like module A2 has the smallest number of large classes in both categories. Lines of code and cyclomatic complexity yield identical results. A1 has the largest classes followed by A3 and finally A2 that has the smallest classes. Surprisingly, the setting is reversed with the LCOM metric. Based on the metric, A2 has the greatest number of non-cohesive classes followed by A1, and A3 has the least of non-cohesive classes. Unfortunately, we cannot offer an explanation for this result.

The reason why module A1 has many large classes when measured by the number of variables is that there are many GUI classes. We can accept slightly larger GUI classes than regular classes and we might be willing to accept that a GUI class can be up to three times larger in terms of variables than a model class as suggested by Lorenz and Kidd (1994). Still, we can see that module A1 has the largest classes, because 23% of its classes have 40 or more variables, whereas in the other two modules only 7.3 and 9.7% of the classes have ten or more variables.

Table 10 Large class measures

Metric data			
Property	Module		
	A1	A2	A3
Number of classes	126	82	31
Classes \geq 10 variables	87 (69.0%)	6 (7.3%)	3 (9.7%)
Classes \geq 20 variables	56 (44.4%)	1 (1.2%)	1 (3.2%)
Classes \geq 40 variables	29 (23.0%)	1 (1.2%)	0 (0.0%)
Classes \geq 30 methods	24 (19.0%)	8 (9.8%)	5 (16.1%)
Classes \geq 50 methods	11 (8.7%)	3 (3.7%)	3 (9.7%)
Classes \geq 100 methods	3 (2.4%)	0 (0.0%)	1 (3.2%)
Classes \geq 500 LOC	32 (25.4%)	13 (15.9%)	5 (16.1%)
Classes \geq 1,000 LOC	12 (9.5%)	4 (4.9%)	2 (6.5%)
Classes \geq 2,000 LOC	6 (5.4%)	1 (1.2%)	1 (3.2%)
Classes \geq 100 CC	34 (27.0%)	12 (14.6%)	5 (16.1%)
Classes \geq 200 CC	16 (12.7%)	4 (4.9%)	3 (9.7%)
Classes \geq 400 CC	9 (7.1%)	2 (2.4%)	2 (6.5%)
Classes \geq 1 LCOM	84 (66.1%)	67 (81.7%)	12 (38.7%)
Classes \geq 10 LCOM	21 (16.6%)	28 (34.1%)	4 (12.9%)
Classes \geq 15 LCOM	8 (6.3%)	15 (18.3%)	1 (3.2%)
Human evaluations (a larger number indicates more of the smell)			
<i>N</i>	5	6	1
Mean	5.20	5.17	2.00
SD	1.095	1.169	–
Median	5	5	2

Means and medians of the smell evaluations under the Human Evaluations section in Table 10 show that modules A1 and A2 are evaluated to contain an equal degree of the Large Class smell. If we compare this to the metric data in Table 10, we can see that the human smell evaluation mean or median does not correlate with the measured number of large classes. The LCOM metric suggests that A2 has more large classes, when all other measures indicate that A1 has more large classes. When we compared the five informants who had evaluated both module A1 and A2, we saw that only one developer had made distinctions between these modules. This developer evaluated in correlation with the majority of the metrics that module A1 contains more of the *Large Class* smell, although the difference on the ordinal scale (1 – 7) was only one. We also studied how the developer who had given the sole evaluation of module A3 had evaluated the other two modules. It appeared that this developer had evaluated modules A1 and A2 with 4 on the seven-point Likert scale, whereas module A3 had received only 2. The developer's evaluations can be in correlation with the metrics, if we study the A1 and A3 modules, but with A2 and A3 the evaluations conflict with all but the LCOM code metric.

5.2. Long Method

The *lines of code* (LOC) metric seems very appropriate for measuring the *Long Method* smell as it effectively measures the method length. However, whether a long method is a problem cannot be simply determined by the lines of code metric because it cannot measure the method's complexity. For example, a long initialization routine with no branches might not pose a problem to a system's evolvability. Therefore, we need to measure complexity as well, and we can do this by using the *cyclomatic complexity* (CC) measure first introduced by McCabe.

An excellent summary covering the existing literature on routine length in terms of lines of code can be found in (McConnell, 2004). According to McConnell, modern programs consist of volumes of short routines with only a few longer ones, but long routines still pose a problem. In McConnell's survey, the recommendation for routine length fluctuates from 50 lines up to 200 lines of code. Lorenz and Kidd (1994) on the other hand suggest a much tighter limit, which is 18 for C++ code. Our case company had recently introduced a coding standard setting a threshold for function length at 40 lines of code. We used the company's threshold of 40 and based on McConnell, we set two other thresholds for method length, 100, and 200 lines of code. Selecting 18 as a threshold would have resulted in 30 to 50% of methods being categorized as long for all modules. Therefore we considered this threshold to be too low for our analysis. For the cyclomatic complexity measure, McCabe originally suggested the limit of 10, and Grady reported that an HP division had determined that 14 should be the maximum cyclomatic complexity in a program. Based on those numbers we set the threshold to 10 and 15 for the cyclomatic complexity of a routine. After we saw the number of methods that exceeded the thresholds, we decided that it would be beneficial to add a third threshold that is higher than the ones introduced. Thus, we added a third threshold of 30. In addition to the threshold values we also report the mean values of both lines of code per method and the cyclomatic complexity per method. However, these cannot be taken as a direct indication of the *Long Method* smell because the averages are greatly affected by the number of small methods.

The metrics related to the method size in the different modules are shown in Table 11 under the metric data section. The data shows that A1 has the longest methods in all expect two occurrences out of the eight. When measured with mean lines of code per method, and the percentage of methods that have 40 lines or more, module A3 has the longest methods. A2 has the shortest methods except when measured with methods that have cyclomatic complexity more or equal to 10 or 15. In those cases A3 has the shortest methods.

The mean and medians of the human evaluations in Table 11 show that module A1 has the most of the Long Method smell. A2 contains slightly less of this smell than A1, and A3 has a minimal amount of this smell according to the evaluations. Notice that for module A1 there is complete agreement between all five evaluators. We can see that there is a correlation for modules A1 and A2, when comparing the smell evaluations and metrics. Comparison between modules A1 and A3 shows that there is correlation between the metrics and human evaluations except when measuring *Long Method* with the average value of LOC per method or with the percentage of methods that contain 40 LOC or more. However, it must be noted that the difference in the evaluations between modules A1 and A3 is too high when compared to their difference in metric values. Finally, smell evaluations for modules A2 and A3 are correlated with the metrics only if we study the method length by methods that have more or equal to 10 or 15 for cyclomatic complexity.

We studied the five informants that had evaluated both modules A1 and A2 and this revealed that two informants had evaluated that modules A1 and A2 have an equal amount of the *Long Method* smell. However, three informants had evaluated—in correlation with the metrics—that A2 has somewhat less (one or two points in the seven point ordinal scale) of the *Long Method* smell. If we leave out the module with only a single evaluation, we can conclude that for the *Long Method* smell the metrics and the smell evaluations are quite satisfactorily correlated.

Table 11 Long method measures

Metric data			
Property	Module		
	A1	A2	A3
Number of methods	2.879	1.105	487
Mean # of LOC / Method	22.28	19.10	26.94
Methods \geq 40 LOC	393 (13.7%)	129 (11.7%)	82 (16.8%)
Methods \geq 100 LOC	105 (3.6%)	21 (1.9%)	13 (2.7%)
Methods \geq 200 LOC	23 (0.8%)	4 (0.4%)	2 (0.4%)
Mean # of CC/Method	5.24	3.71	4.18
Methods \geq 10 CC	356 (12.3%)	100 (9.0%)	38 (7.8%)
Methods \geq 15 CC	212 (7.4%)	48 (4.3%)	19 (3.9%)
Methods \geq 30 CC	79 (2.7%)	12 (1.2%)	7 (1.4%)
Human evaluations (a larger number indicates more of the smell)			
N	5	6	1
Mean	6.00	5.17	2.00
SD	0.000	0.983	–
Median	6.00	5.50	2.00

5.3. Long Parameter List

The *Long Parameter List* smell refers to cases, where a method has too many parameters. We thus need to decide how many is too many. In the era of procedural programming, all data was generally passed as parameters. At that time, the option to passing parameters was to use global data, which was much worse than lengthy parameter lists. McConnell's guidebook for procedural programming (McConnell, 1993) recommends that the *number of parameters* should be limited to seven. Object-oriented programming generally requires less parameter passing, since classes can encapsulate data and operations together. Therefore, we also selected two other parameter limits with values of three and five. We ended up with three opinions on what a long parameter list is. These can be understood as three tolerance levels: low, medium, and high. The maximum number of parameters in these categories is three for low, five for medium, and seven for high.

The metric data in Table 12 shows that the oldest and biggest module (A1) actually has the fewest long parameter lists. Modules A2 and A3 have the same number of long parameter lists in the low and high tolerance groups. In the medium tolerance group, module A2 has more than twice as many long parameter lists. Therefore, it seems that the oldest module seems to be clearly in the best shape, if we measure its internal quality by looking only at this single measure.

When Fowler and Beck (Fowler and Beck, 2000) introduced the *Long Parameter List* smell, they had assumed that long parameter lists are made of primitives rather than objects. This source code material supports that assumption. Of the methods with more than three parameters only 13.9% of parameters are classes, whereas 86.1% are primitives. The maximum number of primitive parameters is 16, whereas the maximum number of class parameters is three. Therefore, it seems clear that the *Long Parameter List* smell mainly consists of primitives.

The Human Evaluations data in Table 12 shows the smell means and medians of the three modules under study. If we compare the two modules with more than one evaluation (A1 and A2), we can see that the informants have evaluated—in

Table 12 Long parameter list measures

Metric Data	Module		
	A1	A2	A3
Number of methods ¹	2 838	1 077	464
Mean # of parameters	1.85	2.05	2.04
Methods \geq 4 parameters	259 (9.1%)	160 (14.9%)	70 (15.1%)
Methods \geq 6 parameters	38 (1.3%)	76 (7.1%)	16 (3.4%)
Methods \geq 8 parameters	4 (0.1%)	13 (1.2%)	5 (1.1%)
Human evaluations (a larger number indicates more of the smell)			
<i>N</i>	5	6	1
Mean	3.40	4.00	1.00
SD	1.140	1.265	–
Median	3.00	4.50	1.00

¹The discrepancy in the number of methods when compared to Table 11 is caused by nested methods. The method count in Table 11 contains the nested methods, but Table 12 does not.

correlation with the metrics—that A2 has the most of the *Long Parameter List* smell. We still have to bear in mind that the standard deviations for the smell means are quite high. The difference between the median of modules A1 and A2 is greater than the mean values. The median values are the ones we wish to look at since the deviation is so large. We also studied the five informants who had evaluated both modules A1 and A2 and found that only one of them had made a difference with the *Long Parameter List* in these two modules. This developer had evaluated—in correlation with the metrics—that module A2 has more of the *Long Parameter List* smell (with Likert scale numbers 5 and 3) whereas others had evaluated that this smell is equally present in both modules.

For the newest module (A3) we received only one smell evaluation. This evaluation is in conflict with the metrics, because it claims that the *Long Parameter List* smell does not exist in the module, whereas the measurements show that module A3 has more of this smell than module A1. It is even more interesting that the developer who had evaluated module A3 also had evaluated module A1 and given it a smell evaluation of three for the *Long Parameter List* smell.

In the case of the *Long Parameter List* smell, the developers assumed—in correlation with the metrics—that module A2 had more long parameter lists than module A1. On the other hand, the opinions of the individual informants were conflicting, since many informants were unable to make distinctions between the two modules that according to the measures contained considerably different amounts of the *Long Parameter List* smell. In addition, the comparison between the individual informants' evaluations showed conflicting results with the metrics.

5.4. Duplicate Code

According to Fowler and Beck (2000), the *Duplicate Code* smell “is number one in the stink parade.” Removing duplication makes programs easier to understand, maintain, and develop further. When we measure the *Duplicate Code* smell, we must decide the size of the duplicated fragments we wish to identify. It is not very wise to remove duplicated code fragments that consist of only a few lines of code, since the effort spent in removing them will outweigh the benefits. We are not aware of any recommendations for how many duplicate code lines are too much. The *same* tool, which was introduced in the beginning of Section 5, by default reports duplicates of ten LOC or more. For us this sounds acceptable, but given that bigger duplicates are more interesting, we also defined thresholds of 15, 20, and 50 lines of code. The amount of duplicate code lines is calculated by summing the redundant code lines, i.e., not including the first occurrence of the code.

The metric data in Table 13 shows the percentages of the duplicate code lines measured. In the table, we can see that module A1 contains the largest chunks of duplicate code. However, module A3 has over 15% of duplicate code if we define the duplicate code chunks to be only ten lines of code or more. If we only measure larger duplicate code chunks, the duplicate code percentage of module A3 drops very quickly. From the source code, we found out that A3 has many methods that terminate in a similar way, i.e., they check out of a critical section, do some exception handling, and then report to the log system that the method has exited. Therefore, this kind of duplication will only cause problems if the exiting sequence has to be changed.

Table 13 Duplicate code measures

Metric data			
Property	Module		
	A1	A2	A3
Total NLOC	83 200	28 654	16 787
Duplicate NLOC ≥ 10	8 576 (10.3%)	1 922 (6.7%)	2 560 (15.2%)
Duplicate NLOC ≥ 15	5 714 (6.9%)	939 (3.3%)	470 (2.8%)
Duplicate NLOC ≥ 20	3 780 (4.5%)	425 (1.5%)	131 (0.8%)
Duplicate NLOC ≥ 50	818 (1.0%)	0 (0.0%)	63 (0.4%)
Human evaluations (a larger number indicates more of the smell)			
<i>N</i>	5	6	1
Mean	3.60	4.00	1.00
SD	1.140	1.265	–
Median	3.00	3.50	1.00

Based on the Human Evaluations data in Table 13 we see that module A2 contains the most of the *Duplicate Code* smell. Although the difference to module A1 is not very big, we can clearly see that the informants' opinions are conflicting with the metrics, because A1 has much more duplicate code according to our measurement. When we looked at the five informants that had evaluated both modules A1 and A2, we saw that two informants had evaluated that A2 contains more duplicate code; two informants had decided that the modules contain the same amount of the *Duplicate Code* smell, and one developer had determined that A1 has more of this smell. To explain why the informants felt that module A2 contained more duplicate code than it actually does, we also tried to look at duplicate code between the modules. We found out that the amount of duplication does not significantly increase in module A2 if we measure its inter-module duplication with A1.

A reason for the differences between the smell evaluations and the source code measurements was revealed to us in a discussion with the developers of the case company. The developers told us that module A2 actually has quite a few lines for which copy–paste coding has been applied, but after each paste operation the code has been modified. The *same* tool is unable to detect this form of duplication. Therefore, the smell evaluations are not as conflicting as they would appear according to the measurements. However, it is a completely different story to find out whether code that has been developed with the copy–paste–modify method is actually duplicate code, and how much modification is needed before the code can no longer be considered duplicate code. However, it is certain that removing identical code chunks is easier than removing slightly modified code.

Again, we also studied the answers of the respondent who had evaluated the newest module (A3) and the other two modules. This developer had evaluated that A2 had more duplicate code smell than A3, whereas in reality they had about the same amount of duplicate code. The result for this single developer is very similar for this smell as for the previous smells.

6. Discussion

In this section, we discuss the results, the answers to our research questions, and compare our findings with related work. Finally, we address the limitations of the study.

6.1. Answers to the Research Questions

Next we answer each research question in Section 3.1 in its own subsection.

6.1.1 Do Software Developers Have a Uniform Opinion on the “Smelliness” of the Source Code?

Unfortunately, due to the small amount of data, we cannot give statistically significant answers to our first research question. However, the standard deviation in the answers is greater than one might expect. Especially for smells like *Switch Statement* and *Inappropriate Intimacy*, the distribution indicates that the informants' opinions were not uniform. We can speculate that perhaps these smells represent problems that are localized to a certain piece of code in the software modules. It is easy to imagine how the usage of switch statements can be concentrated to one location in the code. The same holds for *Inappropriate Intimacy*, which refers to heavy coupling between a pair of classes. It could be concentrated to a cluster of classes. If the developers have focused on working only with certain features in the module and the problems are localized in certain features only, it is sensible to have conflicting opinions. This can explain why developers evaluated these smells differently in the same modules. Generally, the smells that had the most uniform opinions were the smells that had low evaluations for their existence in the software system. This indicates that the developers agree more when certain code problems do not exist, than when there is a great deal of a certain type of problem.

Related work by Shneiderman (1980, pp. 134–138) on subjective code quality evaluation shows similar results. From his work we can see that occasionally the raters had uniform opinions about the code quality and sometimes there was no agreement between the raters. Results from both Shneiderman and us illustrate one of the key problems that involves using human evaluators, namely how to achieve inter-rater agreement. In our study it seems that the lack of uniformity in the smell evaluations can call the use of the smells for internal software quality assessment into question.

6.1.2 Do Demographics of the Developers Affect the Smell Evaluations, and if so, How?

In the study we saw that the lead developers tended to see more structural problems (*Parallel Inheritance Hierarchies*), whereas the regular developers saw more problems on the code level (*Duplicate Code*). This result fits nicely with the idea that the regular developers work closer to the code level and that the lead developers have more design tasks than the regular developers. We observed that the developers with better knowledge of the module evaluated that there is more of the *Lazy Class* smell that is difficult to spot. Again, this could be expected, as the smells that are difficult to spot naturally require better knowledge of the code.

The two developers that had been in the company before the birth of the products and had the longest work experience in the company, tended to evaluate that the software had much less smells than the other ten developers. A possible interpretation is that the two developers have emotional attachment to the software, since they have written a great deal of it and therefore were reluctant to see the smells. Also the fact that it is easier to understand code that you have personally written might affect the evaluations. Another interpretation, suggested by one of the lead developers, is that you get used to the smells. People who have worked with software products for longer periods perhaps understand that complex software products do not always look like textbook examples.

Related work on the differences between novice and expert coders exists, but these studies have mainly focused on the cognitive process and improving the novices' performance, e.g. (Iio et al., 1997; Yu et al., 1994). Thus, we are not aware of any studies where the comparison would have been based on subjective evaluations of the software code quality or maintainability. Also in our study even the least experienced developer had been working in the company as a programmer for 17 months. Consequently, none of our subjects can be really thought as novices. Therefore it seems there is no related work that we could compare these results with.

6.1.3 Do the Developers' Evaluations on Code Smells Correlate with Related Source Code Metrics?

A comparison between the code metrics and the existence of the *Large Class* smell was presented in Section 5.1. Our initial expectation was that the subjective evaluations of the *Large Class* smell would nicely correlate with the measures. We believed that *Large Classes* are quite easy to recognize and recall, and that measuring class size would be easy, or that at least the different code metrics would produce similar results. Conversely, the LCOM metric led to results different from what one would expect based on the other measures, such as the number of variables, number of methods, and method complexity. If each class size measure is individually compared to the developers' evaluations on the *Large Class* smell, the results are conflicting, because the developers evaluated that modules A1 and A2 have an equal amount of the *Large Class* smell. None of the metrics showed values that were close to each other for modules A1 and A2. However, when there are several ways of measuring class size and the metrics yield conflicting results; one can take an alternative view and use the developers' evaluations as a starting point. This will lead to a conclusion that the *Large Class* smell can be viewed as referring to problems in two distinct areas. Firstly, a class can be considered large when the size of a class measured with traditional size and complexity metrics such as lines of code and the number of instance variables indicates this. Secondly, a class can be considered as an excessively large entity when it is not cohesive, e.g., a class consisting of several unrelated items.

The evaluations on *Long Method* and *Long Parameter List* correlated well with the metrics of the smells studied. This is not surprising, because they are both quite simple and should be very easy smells to spot. With the *Long Parameter List* smell, there really can be no issues on how to measure it, since the number of parameters is the only possible measure. With *Long Method* one can always use cohesion measures for measuring the method size. With *Long Method* the developers who made the evaluations that did not correlate with the code metrics were in the

minority. It must be noted that the company had previously measured their method sizes and it seems likely that this has influenced the developers' opinions, and perhaps shifted them to be more in correlation with the metrics. With the *Long Parameter List* smell we also saw some conflicting evaluations when compared to the metrics. However, in both cases the quite high number of methods in the modules can explain some of these differences.

We saw that evaluations of the *Duplicate Code* smell correlated poorly with the metrics. Based on the data it seems that the evaluations by all but one developer conflict with the metrics when it comes to duplicate code. However, two issues concerning the measurement instrument might have caused the biasing of the data. First, copy–paste–modify programming had been used, which resulted in some nearly duplicate code that the tool could not detect. Another problem is that our tool was only able to detect text-based duplication, when it would be more interesting to study lexical or semantic duplication. Nevertheless, the amount of duplicate code detected and the case company's expressed interest for a duplicate code detection tool indicate that redundant code is an important candidate for automatic detection.

The above discussion is mainly based on the evaluations made of modules A1 and A2 since they had the largest number of evaluators; five and six, respectively. Only a single developer evaluated module A3. Fortunately, the developer was one of the most experienced lead developers, who also evaluated modules A1 and A2, which allows a comparison. In all smell evaluations, the developer had evaluated module A3 as having considerably less smells than modules A1 and A2. However, only the LCOM metric for the *Large Class* smell can be said to support the developer's evaluation, and in all other cases, including all other smell measures for the *Large Class* smell, the developer's evaluation did not correlate with the metrics. Module A3 was the newest module, as it was less than a year old, and the developer who made the evaluations had created it for the most parts. These facts have undoubtedly caused bias to the developer's evaluation of module A3.

Overall, it seems that the developers' evaluations of the smells correlated better with the metrics for smells that are simple and easy to spot, i.e., *Long Method* and *Long Parameter List*. For smells that are more difficult to spot, e.g., *Large Class*, which can be difficult to detect because it can be measured in many different ways and the developers working on individual features do not necessarily pay attention to the class sizes of an established program, and *Duplicate Code*, the evaluations did not correlate with the metrics. All smell evaluations made of module A3 by the single developer did not correlate with the selected metrics. If we assume that the tools used provide reliable metrics, we can question the usefulness of the developers' subjective evaluations at least on the module level. Naturally, the other possible interpretation of this result is that the metrics and the tools used in these measurements are not capable of detecting the smells we tried to study.

In related work, Kafura and Reddy (1987) concluded that the expert evaluations on maintainability were in conformance with the complexity source code metric used to measure the maintainability, but there are three differences compared to our study. First, they collected the subjective evaluations by first showing the metrics results to the evaluators and after that they asked what the evaluators thought of the routines that were most complex. We first asked for the evaluations and then collected the metrics and compared the results. There is a distinct possibility that showing the measurement results to the evaluators first might have affected the

evaluators' opinions in Kafura's and Reddy's study. This assumption is supported in our study, where the *Long Method* smell evaluation results correlated best with the metrics. This result might have been affected by the method size measurement performed by the case company prior to our involvement. Second, their informants evaluated the high-level concept of maintainability, whereas we used code smells which are more precise. Third, their subjective evaluations are based on interviews, whereas we used a questionnaire with a Likert scale, which makes the comparison difficult. Nevertheless, it appears that their results are somewhat different from ours.

Oman et al. (Coleman et al., 1994, 1995; Oman and Hagemester, 1994; Welker et al., 1997) used subjective evaluations, collected with randomly selected questions of the 1989 version of the AFOTEC pamphlet,⁶ to create a maintainability measure based on metrics. In validating their maintainability measure with six industrial systems (Oman and Hagemester, 1994) the researchers also experienced some anomalies as some systems had a lower maintainability measure than what one would have expected based on the subjective ratings. This result is somewhat similar to our results. However, the researchers continued tuning their maintainability measure with numerous empirical studies, and they conclude that improved versions of the maintainability measure have successfully matched the practitioners' subjective evaluations (Welker et al., 1997, p. 134). Thus, it seems that their maintainability measure improved in the later stages of the work, but it is unfortunate that these later evaluations have not been meticulously reported. We feel that the differences in our results can be explained by two factors. Firstly, Oman et al., measured the high-level concept of maintainability whereas we measured more exact code smells. Secondly, at the end they had made many studies in tuning their metric to match the subjective evaluations, and it is therefore quite natural that their metric correlated well with the subjective evaluations.

Kataoka et al. (2002) report that experts' subjective evaluations on the effectiveness of refactorings correlated quite well with improvement in coupling metrics. However, in Kataoka's work there was a contradicting case where refactoring would have improved the metrics, but the expert opinion was that the refactoring would not necessarily be effective. Kataoka's work shows that metrics can give good indication whether a single refactoring in the code would be useful to perform. We, on the other hand, studied how well the developers' evaluations on the degree of code smells correlate with the chosen code-smell metrics at the module level. We feel that these differences in the studies can explain the different results.

There also exists a branch of work that has used subjective evolvability by the developers to tune their metrics-based design flaw or code evolvability tools (Marinescu, 2004; Schwanke and Hanson, 1994). The tools have been based on machine learning techniques, such as genetic algorithms and neural networks. In both studies a small empirical assessment indicated that machine learning tools can be tuned to be relatively good predictors of the developers' evaluations. This indicates that subjective evolvability evaluation can be predicted with a learning tool if it is properly trained.

⁶ The 1989 version of the pamphlet is no longer available, but the more recent edition (AFOTEC, 1996) is available

6.2. Limitations

Although we tried to make our study as reliable as possible, for example by using the instructions presented by Pfleeger and Kitchenham (Kitchenham and Pfleeger, 2002a,b,c,d; Pfleeger and Kitchenham, 2001), the study still has many limitations we must address. First of all, we collected the data with an unsupervised survey. Therefore we had no way of making sure the respondent had actually understood the questions. We tried to compensate this limitation by setting the default answer as the option “I don’t know”. Unsupervised surveys also often suffer from a lack of motivation by the informants, which shows up as a low response rate. We did not, however, experience this, as our response rate was 66.7%.

Another major limitation is that we do not know how the developers studied the modules before answering. However, it seems very likely that the modules were not inspected, but that the developers based their smell evaluations on memory recollection. Still, we must keep in mind that the developers only answered the survey concerning the modules they had mostly worked with. However, as human memory is fallible, the developers’ recollections can be biased, making the answers less reliable. Nevertheless, this problem is also often faced in practice when important decisions are based on recollection rather than on systematic assessment.

The third limitation comes from the number of participants. We had 12 developers, who returned 37 smell evaluations concerning 11 modules. These numbers are small when considering the statistical power of the study. With a larger data set the effect of demographic variables could have been more thoroughly analyzed. However, this limitation is difficult to address in real situations, because in practice most software modules are developed by a small number of individuals, i.e., we cannot get a data set where we would have dozens of people evaluating their collectively developed software module.

Limited information was available on the knowledge that the developers had of the modules they evaluated. We only collected the developers’ own subjective evaluations of their familiarity with each module. In addition, knowing how up-to-date the knowledge was, and for how long each developer had been involved with the module, could have provided additional information that might have helped us interpret the results.

There are a few limitations that are specific to research question 1. In the survey, all smells were evaluated against each module the developer had worked with. This could cause bias on the individual opinions based on the quality of the other modules the developer had worked with. For example, consider the situation illustrated in Fig. 2, where developers A and B have worked with one common module X and also with modules Y and Z. In this case, developers A and B could evaluate module X quite differently based on their experiences with module Y or Z.

When we studied the uniformity of the smell evaluations, we saw that the evaluator affected the smell evaluations more than the module in question. This result, in conjunction with the fact that the developers had different attitudes (positive and negative), indicates the problems that exist in using standard deviation to analyze ordinal scale answers. In many studies like this one, values such as mean and standard deviation are calculated from ordinal scale metrics although this is not statistically valid because an ordinal scale is not an interval scale. Instead, if one wishes to study the uniformity of opinions one can use Friedman’s test and calculate Kendall’s coefficient of concordance (Kendall, 1948; Siegel, 1956). Unfortunately, in

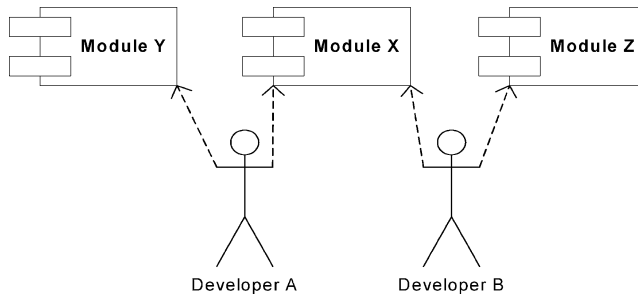


Fig. 2 Two developers with a shared module

this study our data set made it impossible to use these tests, because our informants could not evaluate all the modules, which would have provided a solid base for this type of statistical testing.

There is a limitation that is specific to the second research question. We studied the difference in smell evaluations between the various groups and found that there were significant differences in some smell evaluations between these groups. Since we studied so many smells it is likely, merely by chance, that there were significant differences in the smell evaluations between groups. In studying 23 items we have a 69.3% chance that one or more of these items will be significant at level 0.05 (compute $1 - 0.95^{23}$). This means that when studying the statistically significant differences in any study we must also assess whether the result and its interpretation is sensible or not. We hope that in this study we have been able to explain the significant differences in a way that matches with common sense.

A limitation affecting the third research question relates to the mapping of the smells to the code metrics. We do not know how well the chosen metrics map onto the code smells the developers actually evaluated. Discovering this would have required the developers to explain their evaluations.

Finally, the setting of metric thresholds suffers from a limitation, because for many metrics there exist no empirically grounded thresholds that could have been used as an aid. Therefore, we were forced to set some metric thresholds mostly by using reasoning, common sense, or the distribution percentiles contained in our data set. However, based on the data it seems that the results for the code metrics would have been the same even if the threshold had been slightly different.

7. Conclusions

The purpose of this paper was to study the use of indicators of subjectively perceived code quality, i.e., “bad smells in code.” 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 subjective evolvability evaluation by humans. The empirical research was carried out with a Finnish software product company, whose software products were analyzed. We studied the developers’ subjective smell evaluations and noticed that demographic data

(knowledge, role, work-experience) seemed to explain some of the variances in the smell evaluations. When studying the uniformity of the smell evaluations, we saw how subjective smell evaluations are affected by conflicting perceptions of the different developers. We also applied source code metrics for four smells and compared the results to the subjective smell evaluations. It appears that the developers' evaluations of the more complex smells do not correlate with the source code metrics we used, which indicates that organisations should be careful when using subjective evaluations to evaluate software evolvability. However, based on the results, organisations can also question the correctness and the usefulness of the metrics. For example, for the *Large Class* smell, a composite measure could have been created that would have matched the developers opinions, since the measures themselves were conflicting, as LCOM measure produced different results when compared to other *Large Class* smell measures. Thus, we feel that organisations should not base their software evolvability improvement (refactoring) decisions on either subjective judgment or code metrics alone. Using source code metrics in conjunction with human evaluations is likely to be the best alternative. Code metrics can provide an objective overview of the software, but developers' evaluations and code review is still needed to produce qualitative information about the evolvability of the source code.

To our knowledge there are no similar studies, in which subjective evaluations of code smells would have been studied. Therefore, despite the limitations, we feel that this work offers a contribution to the software engineering community, and provides a basis from which one can learn when building more refined studies.

Based on this initial study, we plan to continue and improve our empirical research on indicators of subjective code quality. We are in the process of using inspections to evaluate the evolvability at the method level. Focusing strictly on the method level will give us better control of this study and will make the study more targeted. The study will also have a far greater number of subjects, thus increasing the statistical power.

In our upcoming study we plan to eliminate the ownership issues when evaluating the source code. However, this study has indicated that ownership could affect the source code evolvability evaluations and this area should be further studied. Another area worth studying is the usage of software design/code metrics in industry. According to our current understanding, such metrics are not widely utilised even though they have been widely researched. The reasons behind this, as well as the pros and cons of using and not using metrics might be particularly interesting.

References

- AFOTEC (1996) Software maintainability evaluation guide. Department of the Air Force, HQ Air Force Operational Test and Evaluation Center
- Arnold RS (1989) Software restructuring. *Proc IEEE* 77:607–617
- Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (2000) Advanced clone-analysis to support object-oriented system refactoring. *Proceedings of Seventh Working Conference on Reverse Engineering*, pp 98–107.
- Bandi RK, Vaishnavi VK, Turk DE (2003) Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans Softw Eng* 29:77–87
- Bansiya J, David CG (2002) A hierarchical model for object-oriented design quality. *IEEE Trans Softw Eng* 28:4–17

- Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J et al (2001) Manifesto for agile software development. [cited 8/21 2003]. Available from <http://agilemanifesto.org/>
- Briand LC, Daly JW, Wüst JK (1997) A unified framework for cohesion measurement in object-oriented systems. Proceedings of the Fourth International Software Metrics Symposium, pp 43–53
- Briand LC, Daly JW, Wüst JK (1999) A unified framework for coupling measurement in object-oriented systems. *IEEE Trans Softw Eng* 25:91–121
- Brown WJ, R Malveau C, McCormick HW, T Mowbray J (1998) *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, New York
- Chidamber SR, Kemerer CF (1994) A metric suite for object oriented design. *IEEE Trans Softw Eng* 20:476–493
- Chidamber SR, Darcy DP, Kemerer CF (1998) Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Trans Softw Eng* 24:629–639
- Chikofsky EJ, Cross JH (1990) Reverse engineering and design recovery: a taxonomy. *IEEE Softw* 7:13–17
- Coleman D, Ash D, Lowther B, Oman PW (1994) Using metrics to evaluate software system maintainability. *Computer* 27:44–49
- Coleman D, Lowther B, Oman PW (1995) The application of software maintainability models in industrial software systems. *J Syst Softw* 29:3–16
- Cusumano MA, Selby RW (1995) *Microsoft secrets*. Free Press, USA
- Cusumano MA, Yoffie DB (1998) Design strategy. In: *Competing on internet time*. Free Press, New York, USA, pp 180–198
- Ducasse S, Rieger M, Demeyer S (1999) A language independent approach for detecting duplicated code. Proceedings of the International Conference on Software Maintenance, Oxford, England, UK, pp 109–118
- Fowler M (2000) *Refactoring: improving the design of existing code*, 1st edn. Addison-Wesley, Boston
- Fowler M, Beck K (2000) Bad smells in code. In: *Refactoring: improving the design of existing code*, 1st edn. Addison-Wesley, Boston, pp 75–88
- Garvin DA (1984) What does “product quality” really mean? *Sloan Manage Rev* 26:25–43
- Grady RB (1994) Successfully applying software metrics. *Computer* 27:18–25
- Halstead MH (1977) *Elements of software science*. Elsevier, New York
- Harrison R, Counsell SJ, Nithi RV (1998) An evaluation of the MOOD set of object-oriented software metrics. *IEEE Trans Softw Eng* 24:491–496
- Henderson-Sellers B (1996) *Object-oriented metrics*. Prentice Hall, Upper Saddle River, New Jersey
- Hitz M, Montazeri B (1996) Chidamber and kemerer’s metrics suite: a measurement theory perspective. *IEEE Trans Softw Eng* 22:267–271
- IEEE (1998) IEEE standard for software maintenance. The Institute of Electrical and Electronics Engineers, Inc, New York
- IEEE (1990) IEEE standard glossary of software engineering terminology. The Institute of Electrical and Electronics Engineers, Inc, New York
- Iio K, Furuyama T, Arai Y (1997) Experimental analysis of the cognitive processes of program maintainers during software maintenance. Proceedings of International Conference on Software Maintenance, pp 242–249
- Kafura DG, Reddy GR (1987) The use of software complexity metrics in software maintenance. *IEEE Trans Softw Eng* 13:335–343
- Kataoka Y, Ernst MD, Griswold WG, Notkin D (2001) Automated support for program refactoring using invariants. Proceedings of International Conference on Software Maintenance, Florence, Italy, pp 736–743
- Kataoka Y, Imai T, Andou H, Fukaya T (2002) A quantitative evaluation of maintainability enhancement by refactoring. Proceedings of the International Conference on Software Maintenance, Montreal, Canada, pp 576–585
- Kendall M, (1948) The problem of m ranking. In: *Rank correlation methods*, 5th edn. Edward Arnold, London, pp 117–143
- Kitchenham BA, Pfleeger SL (1996) Software quality: the elusive target. *IEEE Softw* 13:12–21
- Kitchenham BA, Pfleeger SL (2002a) Principles of survey research part 2: designing a survey. *ACM SIGSOFT Softw Eng Notes* 27:18–20
- Kitchenham BA, Pfleeger SL (2002b) Principles of survey research part 4: questionnaire evaluation. *ACM SIGSOFT Softw Eng Notes* 27:20–23

- Kitchenham BA, Pfleeger SL (2002c) Principles of survey research: part 3: constructing a survey instrument. *ACM SIGSOFT Softw Eng Notes* 27:20–24
- Kitchenham BA, Pfleeger SL (2002d) Principles of survey research: part 5: populations and samples. *ACM SIGSOFT Softw Eng Notes* 27:17–20
- Lehman MM (1980) On understanding laws, evolution, and conservation in the large-program life cycle. *J Syst Softw* 1:213–221
- Li W, Henry SM (1993) Object-oriented metrics that predict maintainability. *J Syst Softw* 23:111–122
- Lorenz M, Kidd J (1994) *Object-oriented software metrics*. Prentice Hall, Upper Saddle River, New Jersey
- Mäntylä MV, Vanhanen J, 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*, pp 381–384
- Marinescu R (2004) Detection strategies: metrics-based rules for detecting design flaws. In: *Proceedings of Software Maintenance, Chicago, Illinois, USA*, pp 350–359
- Maruyama K, Shima K (1999) Automatic method refactoring using weighted dependence graphs. *Proceedings of the International Conference on Software Engineering, Los Angeles, California, USA*, pp 236–245
- McCabe TJ (1976) A complexity measure. *IEEE Trans Softw Eng* 2:308–320
- McConnell S (1993) *Code complete*. Microsoft, Redmond, Washington
- McConnell S (2004) High-quality routines. In: *Code complete 2, 2nd edn*. Microsoft, Redmond, Washington, pp 161–186
- Mens T, Tourne T (2004) A survey of software refactoring. *IEEE Trans Softw Eng* 30:126–139
- Muthanna S, Stacey B, Kontogiannis K, Ponnambalam K (2000) A maintainability model for industrial software systems using design level metrics. *Proceedings of Seventh Working Conference on Reverse Engineering, Brisbane, Australia*, pp 248–256
- Oman PW, Hagemeister J (1994) Constructing and testing of polynomials predicting software maintainability. *J Syst Softw* 24:251–266
- Oman PW, Hagemeister J, Ash D (1991) A definition and taxonomy for software maintainability. *Software Engineering Test Lab, University of Idaho*, pp 91–08
- Pfleeger SL, Kitchenham BA (2001) Principles of survey research. Part 1. Turning lemons into lemonade. *ACM SIGSOFT Softw Eng Notes* 26:16–18
- Pigoski TM (1996) *Practical software maintenance*. Wiley
- Rajlich VT, Bennett KH (2000) A staged model for the software life cycle. *Computer* 33:66–71
- Robillard MP, Coelho W, Murphy GC (2004) How effective developers investigate source code: an exploratory study. *IEEE Trans Softw Eng* 30:889–903
- Rombach DH (1987) Controlled experiment on the impact of software structure on maintainability. *IEEE Trans Softw Eng* 13:344–354
- Schwanke RW, Hanson SJ (1994) Using neural networks to modularize software. *Mach Learn* 15:137–168
- Shepperd MJ (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*. Winthrop, Cambridge, Massachusetts
- Siegel S (1956) *Nonparametric statistics for the behavioral sciences*, 1st edn. McGraw-Hill, New York
- Simon F, Steinbruckner F, Lewerentz C (2001) Metrics based refactoring. *Proceedings Fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal*, pp 30–38
- Sommerville I (2001) *Software engineering*. Addison-Wesley, Reading, Massachusetts
- Stevens W, Myers G, Constantine L (1974) Structured design. *IBM Syst J* 13:115–139
- Subramanyam R, Krishnan MS (2003) Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Trans Softw Eng* 29:297–310
- Succi G, Pedrycz W, Djokic S, Zuliani P, Russo B (2005) An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. *Empirical Software Engineering* 10:81–104
- Sun Microsystems (1999) Code conventions for the java programming language. in Sun Microsystems [database online]. [cited 7/20 1999]. Available from <http://java.sun.com/docs/codeconv/>
- Szulewski PA, Budlong FC (1996) Metrics for ada 95: focus on reliability and maintainability. *CrossTalk—The Journal of Defence Software Engineering* 1996

- Tourwé T, Mens T (2003) Identifying refactoring opportunities using logic meta programming. Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, Benevento, Italy, pp 91–100
- Wake WC (2003) Refactoring workbook, 1st edn. Addison Wesley
- Welker KD, Oman PW, Atkinson GG (1997) Development and application of an automated source code maintainability index. *J Softw Maint Res Pract* 9:127–159
- Yu H, Ikeda M, Mizoguchi R (1994) Helping novice programmers bridge the conceptual gap. Proceedings of International Conference on Expert Systems for Development, Bangkok, Thailand, pp 192–197



Mr. Mika Mäntylä is a researcher and a doctoral student at the Software Business and Engineering Laboratory, Helsinki University of Technology, Finland. He received a Lic. Sc. in 2005 and a M. Sc. in 2003 in software engineering from Helsinki University of Technology. Previously, he was a software developer in the Finnish software industry. His research interests include empirical software engineering, software evolution, code refactoring, and subjective evaluations of code quality.



Mr. Casper Lassenius is a teaching researcher at the Software Business and Engineering Laboratory at Helsinki University of Technology, and the head of the software process research group. He received an M.Sc. in software engineering in 1996 and is currently preparing for his dissertation defense. Mr. Lassenius' research interests include software process modeling, software measurement, software product development, agile development, and globally distributed software development.