# Bad Smells - Humans as Code Critics

Mika V. Mäntylä, Jari Vanhanen, Casper Lassenius
*Helsinki University of Technology*
*Software Business and Engineering Institute*
*P.O. Box 9210, FIN-02015 HUT, Finland*
*{mika.mantyla, jari.vanhanen, casper.lassenius}@hut.fi*

## Abstract

*This paper presents the results of an initial empirical study on the subjective evaluation of bad code smells, which identify poor structures in software. Based on a case study in a Finnish software product company, we make two contributions. First, we studied the evaluator effect when subjectively evaluating the existence of smells in code modules. We found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators. Second, we applied source code metrics for identifying three smells and compared these results to the subjective evaluations. Surprisingly, the metrics and smell evaluations did not correlate.*

## 1. Introduction

Many software related tools, processes, methodologies or techniques claim to improve or assess software quality. Yet, software quality, as quality in general, is largely dependent on one's point of view [15, 24]. Software quality is largely dependent on the quality of the design, e.g., Bansiya and Davis [2] identify software quality attributes, which are mostly effected by software design. The quality attributes they propose are: reusability, flexibility, understandability, extendability, effectiveness, and functionality.

In order to be able to improve the software design, its current state and possible improvement efforts must be made measurable. However, assessing the quality of software design is a difficult task. Widely studied object-oriented metrics [4, 5, 8, 17, 18, 19, 31] offer a way to assess software design quality. Although, some studies report successes in measuring design quality with metrics [2, 7, 9, 30, 38], there are people, especially in the agile community [3], who are critical to using metrics for design quality assessment.

In practice we have noticed – by studying several small and medium sized Finnish software product companies – that design and source code metrics are not widely utilized. Despite this, in our experience, also small companies suffer from harmful effects of software evolution [29], which cause system quality degeneration. Fighting against the harmful effects of software evolution is especially important in software product business, where products evolve in sequential releases. Cusumano and Selby [11] describe how Microsoft uses 20% of its development effort to re-develop the code base of its products. Cusumano and Yoffie [12] 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* [14] to help software developers in recognizing problematic code. These bad smells are supposed to help software developers decide when the software design needs to be improved by refactoring. In most cases, the goal of code refactoring is to make the software easier to understand and/or extend. Fowler and Beck [14] claim that exact criteria cannot be given to determine when software needs refactoring. In their words when it comes to refactoring decision: *"no set of metrics rivals informed human intuition"*. This opinion is in conflict with the idea that the quality of software design can be determined with source code or design metrics. A counterexample to Fowler and Beck is provided by Grady [16], who reports that in some HP divisions there were tight threshold limits that a program was not allowed to exceed, e.g., Fortran programs' cyclomatic complexity [33] should not exceed 14. This threshold was determined based on data from the previous software projects. Also the idea of measuring software maintainability with metrics [9, 10] seems to be conflicting with the idea of assessing maintainability using the quite vaguely defined code smells of Fowler and Beck.

To our knowledge, others have not published studies in which bad smells would have been used as a basis for subjective code evaluation. Development level Anti-Patterns [6], which have some overlap with bad code

Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)
1063-6773/04 $20.00 © 2004 **IEEE**

smells, also seem to be lacking research.

Previously we [32] presented a taxonomy for the bad code smells and also studied the correlation between the existence of the smells. In that study we found out that there was more correlation within the taxonomy's groups than between the groups.

This paper and [32] describe our initial efforts at empirically studying the bad code smells. Thus, despite the very tentative findings of these papers, we hope that they help stimulate more empirical research aiming at critically evaluating, validating and improving our understanding of subjective indicators of design quality.

The rest of the paper is structured in the following way: Section 2 gives an overview of the related work. Section 3 describes the research methods and objectives. In Section 4 we present the results of the study. Section 5 provides the discussion of the results and addresses the limitations of the study. Finally, Section 6 offers the conclusions and future work based on this study.

## 2. Related work

Several studies, e.g., [1, 13, 22, 37, 39] have recently focused on automatically detecting poor design in software. However, there have been only a limited number of studies in which subjective design quality evaluation has been studied or compared with automatic tools. In this section we introduce the relevant prior work we have identified.

Shneiderman et al. (pp. 134-138[36]) reports results from using peer reviews in software code quality evaluation. They conducted three peer-review sessions that each had five professional programmers with similar background and experience as participants. Each programmer provided one of their best programs, which was 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 most). However, in only 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 developers' opinions about the program design, structure, and style that also might explain the results.

Kafura and Reddy [21] studied the relationship between software complexity metrics and software maintainability. Maintainability was measured using system expert evaluations. They conclude that the expert evaluations on maintainability were in conformance with the source code metrics.

Coleman et al. [9, 10] report on the constructions of a maintainability index at Hewlett-Packard (HP). In this study the researchers used source code metrics to create polynomial regression models that measured software maintainability. Then they ordered the maintainability models based on how well they correlated with the subjective evaluations of HP's maintainers. After performing tests on industrial systems, they conclude that the automatic assessment corresponds well to the subjective view of the experts.

Recently, Kataoka et. al [23] report that the subjective evaluation of an expert on the effectiveness of refactorings (design quality enhancement) correlated quite well with improvement in coupling metrics.

Although there are some studies about subjective design/code quality evaluation and the evaluations have been compared with source code metrics, we feel that there is still plenty of research space to be filled. Three out of the four 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 [23] is that the data set consists only on five refactorings and that only one developer evaluated the effectiveness of the refactorings.

## 3. Research methods and objectives

We studied the bad smells using a questionnaire in which we asked software developers to evaluate how much of each bad smell existed in a particular software module. All respondents were software developers in a Finnish software company, which we will refer to as BeachPark. Each respondent answered the questions only regarding the software modules she was familiar with.

### 3.1. Research objectives

Our research objective was to increase our understanding on subjective code quality indicators. In particular, we were interested in the possible variations in the subjective opinions and the explaining factors behind these variations. We also wanted to understand the relationship between subjective code quality evaluations and other code measures. The three research questions that directed our research are as follows:

1. Do, and if so, how, the experience and capability of developers affect the smell evaluations?
2. Do software developers have a uniform opinion on the "smelliness" of the source code?
3. Do the developers' evaluations on code smells correlate with related source code metrics?

### 3.2. Case company

BeachPark is a small Finnish software product development company. The products are not domain-specific.

At the time of the survey, the company employed about 20 software developers. The company had developed two software products in the last 4-5 years and during that time some parts of the products had grown quite complex. 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 1. The modules starting with "A" and "B" are product specific, the others starting with "Y" are shared between the two products.

**Table 1. The modules**

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

*Employee estimates

### 3.3. Smell questionnaire

We collected the data using a two-part web-based questionnaire. In the first part, each respondent provided background information, including age, location (the organization had software developers at two physical locations), 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 part 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, totaling about 35 words. The assessments were made on a seven-point numeric Likert scale, with 1 indicating a total lack of the smell, and 7 indicating large 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".

The population consisted of the software developers of the case company. The response rate was 67% - 12 out of 18 developers working in 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 3, varying from 2 to 6. The number of evaluations per module fluctuated from 1 to 6 and the average was 3,7.

### 3.4. Introducing the smells

We selected three smells for automatic code analysis: Large Class, Long Parameter List, and Duplicate Code. These smells were selected because we thought their operationalization would be quite straightforward, and we had suitable tools to measure them. A more thorough description than the one given here, as well as descriptions of the other code smells can be found in [14].

*Large Class* means a class that is trying to do too much. It has too many responsibilities, which can show up as a large number of instance variables and methods. Large classes are, according to Fowler and Beck, *"prime breeding ground for duplicate code, chaos, and death"*.

*Long Parameter List* means that a subroutine takes too many parameters. Long parameter lists are prone to continuous change, difficult to use, and hard to understand. Therefore, in object-oriented programming one should use objects instead of passing a large number of parameters.

*Duplicate Code* means that software has source code, which appears more than once. Duplicate code makes changing the software more difficult, because the same changes have to be applied in several places. Duplicate code should be unified to make the software more understandable and easier to extend.

The above definitions follow the original idea of the smells, where exact measures or definitions for the smells are lacking. We operationalize the smells later in the paper, when we compare the smell evaluations to the source code metrics.

In addition to the smells introduced above the questionnaire contained questions concerning several other smells. Unfortunately, we could not find a feasible way of measuring those smells using automatic code analysis. Therefore, for those smells, we were restricted to evaluate only the questionnaire data. These smells are introduced next.

*Lazy Class* is a class that is not doing enough – one needs to remove it or increase its responsibility. *Middle Man* refers to a class that is relaying messages without containing significant other functionality. The *Feature Envy* smell exists when a method is too interested in other classes, i.e., the method has excessive coupling. *Long Method* is a method that is too long and tries to do too much on its own. *Switch Statements* refers to cases where type codes or runtime class type detection are used instead of inheritance. *Parallel Inheritance Hierarchies* means that there exist two parallel class hierarchies that are always extended simultaneously.

## 4. Results

This section describes findings of our research. First, we study the human effect in the smell evaluations. Second, we compare the applied source code metrics against the smell evaluations.

### 4.1. Human factors

One key assumption of code smells is that there are no exact conditions, which tell when developers should improve the software design. Instead, individuals should be able to make the decision using their own intuition. As software developers inherently have different predilections it is unlikely that all developers would see the code smells equally. The "curly brace wars" are an example of a tiny programming detail, regarding which – in our experience – developers frequently have conflicting opinions. Therefore, we find it probable that different developers also view the code smells differently.

**4.1.1. Effect of demographics.** We studied how demographic variables like knowledge[1], role, and work experience affected the developers' smell evaluations. We noticed that lead-developers tended to see more structural problems like parallel inheritance hierarchies, but regular developers saw more dead and duplicate code. When we studied how knowledge of a module affected the smell evaluations, we observed that developers with better knowledge of the module reported higher degrees of difficult to spot smells such as Lazy Class and Middle Man.

It is also quite interesting that the two developers with the longest work experience in the case company evaluated that the software had significantly less smells than the other 10 developers. In Table 2 we can see that there is a difference in the overall smell mean and that the p-value from the t-test is significant. The smells that the two most experienced developers evaluated higher than the rest are also listed in the table. These two developers had been with the company for almost eight years. They were also the only developers who had been in the house for the entire lifetime of the software products.

**Table 2. The smell mean differences and p-values from t-test between the most experienced developers and the rest**

| Smell | Mean difference | p-value |
|---|---|---|
| Overall smell mean* | 0,6103 | 0,011 |
| Large Class | 1,42 | 0,004 |
| Long Parameter List | 1,09 | 0,021 |
| Feature Envy | 0,98 | 0,015 |

* Mean from evaluations on all modules and all smells

[1] The respondents' familiarity with the software module was measured using self-assessment on a seven-point Likert scale

**4.1.2. Uniformity of the smell evaluations.** Finally, we studied how uniform the smell evaluations were in the different software modules. We did this by calculating the standard deviations for all smells in each software module. For every smell there was a module that had a very low standard deviation, meaning that developers' opinions about the smell in that module are uniform. On the other hand, the smell evaluations for some modules contained very high standard deviations, e.g., for the Switch Statements smell we measured a standard deviation of 3,06 in one module. In this case, we cannot say anything about how much the module really contains this smell, because the developers' opinions are so conflicting.

We also calculated the standard deviations for each smell evaluated by the same each developer. For example if a developer had evaluated the existence of the Long Method smell in three software modules, we calculated the standard deviation for these three evaluations. At first, this might sound pointless, because we are calculating the mean of the evaluations of different software modules. However, when we compared these standard deviations against the standard deviations of individual modules, we saw that the deviations were lower for developers than they were for modules. This can be interpreted as indicating that the individual doing the evaluation affected the results more than the actual module.

We also studied the differences between developers' attitudes regarding the amount of smells. We compared five developers that all had evaluated two particular modules. Based on that we noticed that one developer clearly had the most positive attitude towards the quality of the code. 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. Therefore it seems that the developers' different attitudes also can explain the difference in smell evaluations.

### 4.2. Source code metrics vs. human evaluation

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

We used an enhanced version of a tool called *same*[2] to measure the number of duplicate code lines. Technically, *same* does not measure duplicate code lines, since it only

[2] http://sourceforge.net/projects/same

investigates whether the lines are lexically identical after removal of white spaces. This means that *same* will not recognize all possible duplicated lines, e.g., if variables have different names. To gather measures for the Large Class and Long Parameter List smells we used a tool called *SDMetrics*[3].

**4.2.1. Large class.** The first question arising when working with the Large Class smell is, what is exactly a large class. Fowler and Beck [14] say that a large class can often be spotted by looking at the number of instance variables. 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 (pp. 73-84[6]) points out that number of operations and variables is good measure for such anti-pattern. The number of operations in a class is also a metric whose extended version Chidamber and Kemerer used in their metrics suite [8]. The number of operations measure was used in object-oriented design quality assessment by Bansiya and David [2]. In our opinion class cohesion would be a good measure for the Large Class smell, because large classes often try to do too many things. Unfortunately, we did not have suitable tools to collect such a measure. Therefore, we limited the study and only measured the number of variables/attributes and operations/methods.

We also needed some exact thresholds for the Large class smell. The Anti-Pattern book [6] 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 operations often indicates the presence of the Blob. Much tighter thresholds are presented by Lorenz and Kidd [31], 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 operations and a UI class should have a maximum of 40 operations. However, as most studies do not provide specific thresholds for class size and because they would be language specific anyway, we have decided not to use any given value as such.

Based on the discussion above we have created two categories to measure the Large Class smell. One is based on attributes and the other is based on operations. With variables and operations we have three limits for a Large Class. With variables our limits are 10, 20, and 40 or more variables, and with operations the limits are 30, 50, and 100 or more.

The metrics related to the class size in different modules are in Table 3 under the metric data section. The data shows that the A1 module clearly has the largest classes, measured by the number of variables. When measuring class size with the number of operations, 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 among these modules. The reason why the A1 module has many large classes is that many classes in that module are GUI classes. We can accept slightly larger GUI classes than regular classes and we might be willing to accept that GUI class can be up to three times larger in terms of variables than a model class as suggested by Lorenz and Kidd [31]. Still we can see that module A1 has the largest classes, because 23% of its classes have 40 or more variables, while in the other two modules 7,3% and 9,7% of classes have 10 or more variables.

**Table 3. Large Class measures**

| **Metric Data** | | | |
|---|---|---|---|
| **Property** | **Module** | | |
| | **A1** | **A2** | **A3** |
| Number of classes | 126 | 82 | 31 |
| Classes ≥30 oper. | 24 (19,0%) | 8 (9,8%) | 5 (16,1%) |
| Classes ≥50 oper. | 11 (8,7%) | 3 (3,7%) | 3 (9,7%) |
| Classes ≥100 oper. | 3 (2,4%) | 0 (0,0%) | 1 (3,2%) |
| Classes ≥10 variables | 87 (69,0%) | 6 (7,3%) | 3 (9,7%) |
| Classes ≥20 variables | 56 (44,4%) | 1 (1,2%) | 1 (3,2%) |
| Classes ≥40 variables | 29 (23,0%) | 1 (1,2%) | 0 (0,0%) |
| **Human Evaluations** | | | |
| N | 5 | 6 | 1 |
| Mean | 5,20 | 5,17 | 2,00 |
| SD | 1,095 | 1,169 | - |
| Median | 5 | 5 | 2 |

Smell means and medians of the smell evaluations in Table 3 under the Human Evaluations section show that modules A1 and A2 contain an equal amount of the Large Class smell. If we compare this to the metric data in Table 3, we can see that the smell mean or median do not correlate with the measured number of large classes. When we compared the five developers who had evaluated both module A1 and A2, we saw that only one developer had made distinctions between these modules. This developer had evaluated in correlation that module A1 contains more of the Large Class smell, although the difference on the Likert scale (1-7) was the smallest possible. We also studied how the developer who had given the sole evaluation on 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, while module A3 had received only 2. The developers' evaluations can be correct compared to metrics, if we study the A1 and A3 modules, but with A2 and A3 the evaluations conflict with the source code measures.

**4.2.2. Long Parameter List.** As mentioned earlier, 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 pro-

[3] http://www.sdmetrics.com/

gramming, 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 [34] 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 thus have ended up with three opinions on what a long parameter list is. The 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 4 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. It is quite interesting that the oldest module seems to be clearly in the best shape, if we measure its internal quality by just looking at this single measure.

When Fowler and Beck [14] 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. From methods with over three parameters only 13,9% of parameters are classes, while 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.

**Table 4. Long Parameter List measures**

| **Metric Data** | | | |
|---|---|---|---|
| **Property** | **Module** | | |
| | **A1** | **A2** | **A3** |
| Number of methods | 2838 | 1077 | 464 |
| Mean # of param. | 1,85 | 2,05 | 2,04 |
| Methods ≥4 param. | 259 (9,1%) | 160 (14,9%) | 70 (15,1%) |
| Methods ≥6 param. | 38 (1,3%) | 76 (7,1%) | 16 (3,4%) |
| Methods ≥8 param. | 4 (0,1%) | 13 (1,2%) | 5 (1,1%) |
| **Human Evaluations** | | | |
| N | 5 | 6 | 1 |
| Mean | 3,40 | 4,00 | 1,00 |
| SD | 1,140 | 1,265 | - |
| Median | 3,00 | 4,50 | 1,00 |

The Human Evaluations data in Table 4 shows the smell means and medians of the three modules under study. If we compare the two modules with more than one evaluation, we can see that developers have evaluated - in 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 and with different sampling the results might look different. The difference between the Long Parameter smell median of modules A1 and A2 is greater than the smell mean values. The median values are the ones we wish to look at since the deviation is so large. We also studied the five developers 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 Lists smell (with Likert scale numbers 5 and 3) while others had evaluated that this smell is equally present in both of the 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 exists in the module, whereas the measurement 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, 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 individual developers' were untrustworthy, since many developers were unable to make distinctions between two modules, which according to measures contained different amount of smells. In addition, the comparison between the individual developer's evaluations showed conflicting results with the metrics.

**4.2.3. Duplicate Code.** According to Fowler and Beck [14], 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 what the size of the duplicated fragments we wish to identify. It is not very wise to remove duplicated code fragments that consist of only few lines of code, since the effort spent in removing them will outweigh the benefits. We are not aware of any recommendations on how many duplicate code lines are too much. The *same* tool, which was introduced in the beginning of section 4.2, by default reports duplicates of 10 LOC or more. For us this sounds acceptable, but given that bigger duplicates are more interesting, we also defined groups with 15, 20, and 50 lines of code. The line of code in this context is actually NLOC, which counts only code lines and ignores empty lines and comment lines. 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 5 shows the percentages of

duplicate code lines measured in NLOC. In the table, we can see that module A1 contains the largest chucks of duplicate code. However, module A3 has over 15% of duplicate code if we define the duplicate code chunks to be only 10 NLOC 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 the 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 5. Duplicate Code measures**

| **Metric data** | | | |
|---|---|---|---|
| **Property** | **Module** | | |
| | **A1** | **A2** | **A3** |
| Total NLOC | 83200 | 28654 | 16787 |
| Duplicate NLOC ≥10 | 8576 (10,3%) | 1922 (6,7%) | 2560 (15,2%) |
| Duplicate NLOC ≥15 | 5714 (6,9%) | 939 (3,3%) | 470 (2,8%) |
| Duplicate NLOC ≥20 | 3780 (4,5%) | 425 (1,5%) | 131 (0,8%) |
| Duplicate NLOC ≥50 | 818 (1,0%) | 0 (0,0%) | 63 (0,4%) |
| **Human Evaluations** | | | |
| N | 5 | 6 | 1 |
| Mean | 3,60 | 4,00 | 1,00 |
| SD | 1,140 | 1,265 | - |
| Median | 3,00 | 3,50 | 1,00 |

Based upon the Human Evaluations data in Table 5 we can notice 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 developers are mistaken, because module A1 has much more duplicate code according to our measurement. When we looked at the five developers that had evaluated both modules A1 and A2, we saw that two developers had evaluated that A2 contains more duplicate code, two developers had decided that the modules contain the same amount of Duplicate Code smell, and one developer had determined that A1 has more of this smell. To explain why developers felt that module A2 contained more Duplicate Code than it actually does, we also tried to look at duplicates between 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 mean and the source code measurements according to the case company developers is 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 slightly 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[4].

Again, we also studied the answers of the respondent who had evaluated the newest module, A3, and the other two modules as well. 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 as for the previously compared smells.

## 5. Discussion

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

### 5.1. Answers to the research questions

**5.1.1. Do, and if so, how, the experience and capability of developers affect the smell evaluations?** In the study we saw that lead-developers tended to see more structural problems, whereas regular developers saw more problems on the code level. This result fits nicely with the idea that regular developers work closer to the code level and that lead developers have more design tasks than regular developers. We observed that developers with better knowledge of the module evaluated that there are more smells that are difficult to spot like Lazy Class and Middle Man. Again this could be expected, as the smells that are difficult to spot naturally require better understanding of the code.

Also the two developers, which had been with 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 10 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 are 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. Maybe people who have worked with software products for longer periods understand that complex software products do not always look like a textbook example.

There has been work on differences between novice

[4] It is a completely different story to find out whether code that has been developed with a 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.

and expert coders, but it has mainly focused on the cognitive process and how to improve the novices' performance, e.g. [20, 40]. Thus, we are not aware of any studies where the comparison would have been based on subjective evaluations of software code quality or maintainability. Also in our study even the least experienced developer had been working as a programmer with the company for 17 months, consequently none of our subjects can be really thought as novices. Therefore there seems to be no related work that we could compare our results with.

**5.1.2. Do software developers have a uniform opinion on the "smelliness" of the source code?** From the data we saw that for every smell we could find a module where the smell evaluations were uniform. On the other hand when we studied the uniformity of the smell evaluations, we saw that the evaluator effected the smell evaluations more than the module in question. Part of this unexpected bias could be a result of the setup of the survey. In the survey, all smells were evaluated against each module the developer had worked with. This could cause bias on individual opinions based on the quality of the other modules the developer had worked with. For example, consider the situation illustrated in Figure 1, where developers A and B have worked with one common module X and also with modules Y and Z. Thus, in this case developers A and B could evaluate module X quite differently based on their experiences with module Y or Z.

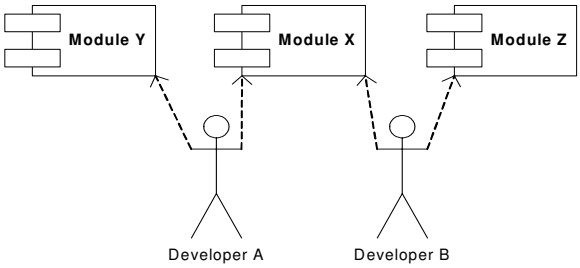


**Figure 1. Two developers with a shared module**

We also saw how developer attitudes regarding the amount of smells effected the smell evaluations. We studied five developers that all had evaluated the same two modules. We found that one developer clearly had a positive attitude towards the two modules (low smell evaluations) while other had the most negative (high smell evaluations).

Related work by Shneiderman (pp. 134-138[36]) from subjective code quality evaluation shows similar results. From his work we can see that at some cases raters had uniform opinions about code quality and in others there was no agreement between the raters. These results by Shneiderman and us illustrate one of the key problems that involves using human evaluators, which is how to achieve inter-rater agreement. In our study it seems that lack of uniformity in the smell evaluations questions the use of the smells for internal software quality assessment.

**5.1.3 Do the developers' evaluations on code smells correlate with related source code metrics?** Comparison with metrics and the *Large Class* smell was presented in Section 4.2.1. Our initial expectation was that the subjective evaluations of the Large Class smell would nicely correlate with the measures, because we believed that Large Classes are quite easy to spot. However, we feel that our data shows that developers' evaluations on the Large Class smell seem to be conflicting, when compared to large class measures. It was unfortunate that we did not have class cohesion measures available that could have provided more insight in detecting large classes.

The evaluations on *Long Parameter List* correlated best with metrics when compared to the other two smells we studied in detail. This is not surprising, because Long Parameter List should be a very easy smell to spot. Also with Long Parameter List there really can be no issues on how to measure it, since the number of parameters is the only possible measure. However, even with this smell we saw some conflicting evaluations when compared to metrics. The quite high number of methods in the modules can also explain some of these differences.

We saw how evaluations on the *Duplicate Code* smell were the most uncorrelated with the metrics. Based on the data it seems that the evaluations by all but one developer conflict with the measurements when it comes to duplicate code. However, two issues concerning the measurement instrument might have caused biasing of the data. First, copy-paste-modify programming had been used, which resulted some nearly duplicate code that the tool could not detect. Another problem is that our tool was only able to detect lexical duplication, when it would be more interesting to study semantic duplication. This issue is, however, a fundamental problem with tool based duplicate code detection, since automatically detecting semantic duplication is impossible. Nevertheless, the amount of duplicate code detected and the also the case-companies expressed interest to duplicate code detection tool indicates that redundant code is important candidate for automatic detection.

Overall it appears that developers' evaluations of the smells do not correlate with the used source code metrics. If we assume that the tools used provide reliable metrics, we can question the usefulness of the developers' subjective evaluations at least at the module level. Naturally, the other possible interpretation of this result is that the metrics and tools used in these measurements are not capable of detecting the smells we tried to study.

In related work Kafura and Reddy [21] conclude that the expert evaluations on maintainability were in conformance with the complexity source code metric used to measure the maintainability. On the other hand their informants evaluated the higher-level concept of maintain-

ability, when we used code smells, which are more precise. Also, their subjective evaluations are based on interviews, when we used a questionnaire with a Likert scale, which makes the comparison a bit more difficult. Nevertheless, it appears that their results are somewhat different from ours.

Coleman et al. [9, 10] used subjective evaluations to create a metrics based maintainability measure. It is therefore quite natural that their metric correlated well with subjective evaluations. Kataoka et al. [23] report that the subjective evaluation of an expert on the effectiveness of refactorings correlated quite well with improvement in coupling metrics. Although in Katoaka's work there was also a case where refactoring would have improved the metrics, but the expert opinion was that the refactoring would not necessarily be effective. Katoaka's work tells 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 developer evaluations on the amount of code smells correlate with the chosen code-smell metrics at module level. We feel that these differences in the studies also can explain the different results.

### 5.2. Limitations

Although we tried to make our study as reliable as possible, for example by using the instructions presented by Pfleeger and Kitchenham [25, 26, 27, 28, 35], the study still has at least three limitations we must address. First, 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 to "I don't know"-option. Unsupervised surveys also often suffer from lack of motivation by the informants, which shows up in low response rate. However, we did not experience this, as our response rate was 66,7%.

The second major limitation is that we don't 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. As human memory is fallible, the developers' recollections can be biased, which makes the answers less reliable. However, the same problem is faced often also in practice when important decisions are based on recollection or gut feeling rather than systematic assessment.

The third limitation comes from the number of participants. We had 12 developers, which returned 37 smell evaluations concerning 10 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 hard to address in real situations, because in practice most software modules are developed by small number of individuals i.e. we cannot get a data set were we would have dozens of people evaluating their collectively developed software module.

## 6. Conclusion

The purpose of the empirical research reported in this paper was to study the use of indicators of subjectively perceived code quality, so called "bad smells". The research was carried out with a Finnish software product company, whose software products were analyzed. We studied the subjective smell evaluations of developers and noticed that demographic data (knowledge, role, work-experience) seemed to explain some of the variances in smell evaluations. Also, when we studied the uniformity of the smell evaluations, we saw how the subjective smell evaluations are affected by conflicting perceptions of different developers. We also applied source code metrics for three smells and compared the results to the subjective smell evaluations. It appears that developers' evaluations of the smells do not correlate with the used source code metrics.

To our knowledge there are no similar studies, in which the perceived evaluations of bad smells would have been studied. Therefore despite the limitations, we feel that this work offers a novel contribution to the software engineering community, and provides a basis upon which further, more refined studies can be built.

Based on this initial study, we plan to continue and improve our empirical research on indicator of perceived code quality. We are in the process of using inspections to evaluate the "smelliness" at the method level. This study also has a far greater amount of subjects thus increasing the statistical power.

Another area worth studying in industry is the usage of software design/code metrics. According to our current understanding, such metrics are not widely utilized even though they have been widely researched. The reasons behind this fact, as well as pros can cons on using and not using metrics might be particularly interesting.

## References

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

[2] J. Bansiya and C.G. David, "A Hierarchical Model for Object-Oriented Design Quality," *IEEE Trans. Software Eng.,* vol. 28, no. 1, pp. 4-17, 2002.

[3] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andy Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert Martin C., Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas. , "Manifesto for Agile Software Development," 2001, http://agilemanifesto.org/

[4] L.C. Briand, J.W. Daly and J.K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems,"

*IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 91-121, 1999.

[5] L.C. Briand, J.W. Daly and J.K. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," in Proceedings of the Fourth International Software Metrics Symposium, 1997, pp. 43-53.

[6] W.J. Brown, R. Malveau C., H.W. McCormick and T. Mowbray J., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis,* New York: Wiley, 1998.

[7] S.R. Chidamber, D.P. Darcy and C.F. Kemerer, "Managerial use of metrics for object-oriented software: an exploratory analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 629-639, 1998.

[8] S.R. Chidamber and C.F. Kemerer, "A Metric Suite for Object Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 476-493, 1994.

[9] D. Coleman, D. Ash, B. Lowther and P.W. Oman, "Using Metrics to Evaluate Software System Maintainability," *Computer,* vol. 27, no. 8, pp. 44-49, 1994.

[10] D. Coleman, B. Lowther and P.W. Oman, "The Application of Software Maintainability Models in Industrial Software Systems," *J.Syst.Software,* vol. 29, no. 1, pp. 3-16, 1995.

[11] M.A. Cusumano and R.W. Selby, *Microsoft Secrets,* USA: The Free Press, 1995.

[12] M.A. Cusumano and D.B. Yoffie, "Design Strategy," in *Competing on Internet Time,* New York, USA: The Free Press, 1998, pp. 180-198.

[13] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code," in Proceedings of the International Conference on Software Maintenance, 1999, pp. 109-118.

[14] M. Fowler and K. Beck, "Bad Smells in Code," in *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 2000, pp. 75-88.

[15] D.A. Garvin, "What does "product quality" really mean?" *Sloan Management Review,* vol. 26, no. 1, pp. 25-43, Fall. 1984.

[16] R.B. Grady, "Successfully Applying Software Metrics," *Computer,* vol. 27, no. 9, pp. 18-25, September. 1994.

[17] R. Harrison, S.J. Counsell and R.V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 491-496, 1998.

[18] B. Henderson-Sellers, *Object-Oriented Metrics,* Upper Saddle River, New Jersey, USA: Prentice Hall, 1996.

[19] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metrics suite: a measurement theory perspective," *IEEE Trans. Software Eng.,* vol. 22, no. 4, pp. 267-271, 1996.

[20] K. Iio, T. Furuyama and Y. Arai, "Experimental analysis of the cognitive processes of program maintainers during software maintenance," in Proceedings of International Conference on Software Maintenance. 1997, pp. 242-249.

[21] D. Kafura G. and G. Reddy R., "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Trans. Software Eng.,* vol. 13, no. 3, pp. 335-343, 1987.

[22] Y. Kataoka, M.D. Ernst, W.G. Griswold and D. Notkin, "Automated support for program refactoring using invariants," in Proceedings of International Conference on Software Maintenance, 2001, pp. 736-743.

[23] Y. Kataoka, T. Imai, H. Andou and T. Fukaya, "A Quantative Evaluation of Maintainability Enhancement by Refactoring," in Proceedings of the International Conference on Software Maintenance, 2002, pp. 576-585.

[24] B.A. Kitchenham and S.L. Pfleeger, "Software Quality: The Elusive Target," *IEEE Software,* vol. 13, no. 1, pp. 12-21, 1996.

[25] B.A. Kitchenham and S.L. Pfleeger, "Principles of survey research part 2: designing a survey," *ACM SIGSOFT Software Engineering Notes,* vol. 27, no. 1, pp. 18-20, 2002.

[26] B.A. Kitchenham and S.L. Pfleeger, "Principles of survey research: part 3: constructing a survey instrument," *ACM SIGSOFT Software Engineering Notes,* vol. 27, no. 2, pp. 20-24, 2002.

[27] B.A. Kitchenham and S.L. Pfleeger, "Principles of survey research part 4: questionnaire evaluation," *ACM SIGSOFT Software Engineering Notes,* vol. 27, no. 3, pp. 20-23, 2002.

[28] B A. Kitchenham and S.L. Pfleeger, "Principles of survey research: part 5: populations and samples," *ACM SIGSOFT Software Engineering Notes,* vol. 27, no. 5, pp. 17-20, 2002.

[29] M.M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," *The Journal of Systems and Software,* vol. 1, pp. 213-221, 1980.

[30] W. Li and S.M. Henry, "Object-Oriented Metrics that Predict Maintainability," *J.Syst.Software,* vol. 23, no. 2, pp. 111-122, 1993.

[31] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics,* Upper Saddle River, New Jersey, USA: Prentice Hall, 1994.

[32] M.V. Mäntylä, J. Vanhanen and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in Proceedings of the International Conference on Software Maintenance, 2003, pp. 381-384.

[33] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 4, pp. 308-320, 1976.

[34] S. McConnell, *Code Complete,* Redmond, Washington, USA: Microsoft Press, 1993.

[35] S.L. Pfleeger and B.A. Kitchenham, "Principles of survey research: part 1: turning lemons into lemonade," *ACM SIGSOFT Software Engineering Notes,* vol. 26, no. 6, pp. 16-18, 2001.

[36] B. Shneiderman, *Software Psychology: Human factors in Computer and Information Systems,* Cambridge, Massachusetts, USA: Winthrop Publishers, 1980.

[37] F. Simon, F. Steinbruckner and C. Lewerentz, "Metrics based refactoring," in Proceedings Fifth European Conference on Software Maintenance and Reengineering, 2001, pp. 30-38.

[38] R. Subramanyam and M.S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Trans. Software Eng.,* vol. 29, no. 4, pp. 297-310, 2003.

[39] T. Touwré and T. Mens, "Identifying refactoring opportunities using logic meta programming," in Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, 2003, pp. 91-100.

[40] Yu He, M. Ikeda and R. Mizoguchi, "Helping novice programmers bridge the conceptual gap," in Proceedings of International Conference on Expert Systems for Development, 1994, pp. 192-197.