Code smells as system-level indicators of maintainability: An Empirical Study

Aiko Yamashita[a,b], Steve Counsell[c]

[a]*Simula Research Laboratory, P.O. Box 134, Lysaker, Norway*
[b]*Dept. of Informatics, University of Oslo, Oslo, Norway*
[c]*Brunel University, Kingston Lane, Uxbridge, Middlesex, UK*

Abstract

Code smells are manifestations of design flaws that can degrade code maintainability if left to fester. The research in this paper investigates the potential of code smells to reflect system-level indicators of maintainability. We report a study where the strengths and limitations of code smells are evaluated against existing evaluation approaches. We evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment and the Chidamber and Kemerer suite of metrics. The systems were maintained over a period up to 4 weeks. During maintenance, *effort* (person-hours) and *number of defects* were measured, to validate the different evaluation approaches. Results suggest that code smells are strongly influenced by size. An implication is that code smells are likely to yield inaccurate results when comparing the maintainability of systems differing in size. When comparing the evaluation approaches, expert judgment was found as the most accurate and flexible since it considered effects due to a system's size and complexity and could adapt to different maintenance scenarios. We also found that code smells complemented expert-based evaluation, since they can identify critical code that experts can sometimes overlook.

Keywords: code smells, maintainability, empirical study, system evaluation

1. Introduction

Numerous studies have reported that significant effort/cost in software projects is allocated to maintenance (Bennett, 1990; Harrison and Cook, 1990; Abran and Nguyenkim, 1991; Pigoski, 1996; Jones, 1998). Consequently, it becomes important to develop strategies for evaluating the maintainability of a system. Most known

Email addresses: `aiko@simula.no` (Aiko Yamashita), `steve.counsell@brunel.ac.uk` (Steve Counsell) maintainability assessments are based on software measures, such as the *Maintainability Index* (Welker, 2001) and the object-oriented suite of metrics proposed by Chidamber and Kemerer (C&K) (1994).

Code smells reflect code that 'screams out' to be refactored (Fowler, 1999) and can degrade aspects of code quality such as understandability and changeability; they can also potentially lead to the introduction of faults. Fowler and Beck (1999) provided a set of informal descriptions for twenty-two code smells and associated them with different refactoring strategies that can be applied to remedy those smells. The main motivation for using code smells for system maintainability assessment is that they constitute software features that are potentially easier to interpret than traditional OO software measures. They can pinpoint problematic areas of code and, since many of the descriptions of code smells in (Fowler, 1999) are based on situations that developers face on a daily basis, they are potentially easier to understand and address by developers.

The research conducted by Anda (2007) evaluated and compared the maintainability of four Java applications with nearly identical functionality but different design and implementation through software measures (*i.e.,* C&K code metrics) and expert judgment[1]. The study concluded that software measures and expert judgment addressed different aspects of maintainability in a system, and consequently combining them can lead to more complete evaluations of maintainability. We draw heavily on that study in this paper.

We report on our experiences of using bad smells in code (henceforward just 'code smells') for evaluating and comparing system-level maintainability of the same four systems evaluated in (Anda, 2007) and we compare our results to those previously derived via expert judgment and the C&K metrics. Results from the three evaluation approaches were then compared to empirical measures of maintainability (*i.e.,* maintenance effort and defects) resulting from a maintenance project where change requests were implemented in each of those four systems over a period of up to 4 weeks. Smell density (*i.e.,* Smells/LOC) was used to adjust for size, but this measure conferred too much advantage on the largest system (which had a very low smell density), leading to naïve assessments of maintainability. When removing the largest system from the analysis and comparing the remaining three systems (all three of similar size), smell density could single out the system with the lowest maintainability. As a result, smell density may not be useful when comparing the maintainability of systems differing considerably in size. When comparing code smells with other evaluation approaches, we found

[1]According to Shanteau (1992), in cognitive psychology, *experts* are operationally defined as those who have been recognized within their profession as having the necessary skills and abilities to perform at the highest level.

that structural measures provided more insight into which system had the most "balanced design", but this approach ignored the effect of size when maintenance consisted of small/medium tasks. Expert judgment was the most versatile, since it considered both the effect of size and different maintenance scenarios (*i.e.,* small extensions to a system vis-à-vis large extensions).

The remainder of this paper is structured as follows: In the next section, we provide related work. Section 3 describes the different elements of this case study, namely the systems under analysis, the early evaluations performed by Anda, our approach/rationale to evaluate maintainability via code smells and the maintenance project from which the empirical measures were derived. In Section 4 we present the results of the maintainability evaluation based on code smells. We then describe the outcomes from the maintenance project (Section 5). Section 6 presents the validation and comparison of the three evaluation approaches and Section 7 discusses the results and their validity. Finally, we summarize the study findings and present plans for future work in Section 8.

2. Related Work

2.1. Study Context

In the study conducted by Anda (2007), it was found that software measures namely the C&K code metrics and expert judgment addressed different aspects of maintainability in a system; thus combining them led to more complete evaluations of maintainability. Nevertheless, the path from evaluation to development of concrete action plans is not clear in expert-based evaluations. As Anda points out (2007), if one were to ask an expert to identify the areas of code to modify in order to improve maintainability it would be a hugely time-consuming and expensive process. The same kind of limitation applies to software measures. As Marinescu (2002) and Heitlager (2007) suggest, code metrics lack 'guidelines' to improve their value (and thereby maintainability). Since some code smells can now be detected by automated means, it is appealing to evaluate their potential for indicating the maintainability at the system level and to compare that to existing approaches such as software measures and expert judgment. Our study investigated the same four systems reported in (Anda, 2007) and this enabled us to compare a code smells-based approach with previous evaluation approaches without introducing new systems. For instance, involving new systems would have lead to the need for considering many moderator factors, and contextual variability. We had the opportunity to conduct and observe a maintenance project involving these four systems. This enabled us to validate empirically the different maintainability evaluation approaches and understand better their strengths and limitations when used in realistic settings.

2.2. Code Smells

The concept of a code smell was introduced as an indicator of software design flaws that could potentially affect maintenance. As Fowler (1999) indicates, a code smell is a sub-optimal design choice that can degrade different aspects of code quality such as understandability and changeability. Code smells have become an established concept for identifying patterns or aspects of software design that may cause problems for further development and maintenance of the system (Fowler, 1999; Lanza and Marinescu, 2005; Moha et al., 2010). Van Emden and Moonen (2001) provided the first formalization of code smells and described a tool for Java programs, while Mäntylä et al. (2003) and Wake (2003) both proposed taxonomies for code smells. Travassos et al. (1999) proposed a process based on manual detection to identify code smells for quality evaluations. In (Mäntylä et al., 2004; Mäntylä and Lassenius, 2006) Mäntylä et al. provide an empirical study of subjective detection of code smells and compares that approach with automated metrics-based detection. Results from manual detection were not uniform between experienced developers and novices (*i.e.,* experienced developers reported more complex smells). Further Mäntylä et al. found that developers with less experience with modules identified more code smells than developers familiar with modules. Finally, when comparing subjective detection with automated detection, it was found that developer evaluation of complex code smells did not correlate with the results of the metrics detection. They concluded that subjective evaluations and metric-based detection should be used in combination. Mäntylä also reports on a experiment for evaluating subjective evaluation for code smells detection and refactoring decisions (Mäntylä, 2005). The research observed the highest inter-rater agreements to be between evaluators for simple code smells, but when the subjects were asked to make refactoring decisions, low agreement was observed. Previous studies have investigated the effects of individual code smells on different maintainability related aspects such as *defects* (Monden et al., 2002; Li and Shatnawi, 2007; Juergens et al., 2009; D'Ambros et al., 2010; Rahman et al., 2010), *effort* (Deligiannis et al., 2003, 2004; Lozano and Wermelinger, 2008; Abbes et al., 2011) and *changes* (Kim et al., 2005; Khomh et al., 2009; Olbrich et al., 2010).

D'Ambros et al. (2010) analyzed code in seven open source systems and found that neither 'Feature Envy' nor 'Shotgun Surgery' smells was consistently correlated with defects across systems. Juergens et al. (2009) observed the proportion of inconsistently maintained 'Duplicated Code' smells in relation to the total set of duplicates in C#, Java and Cobol systems and found (with the exception of Cobol) that 18% of inconsistent 'Duplicated Code' smells were positively associated to faults. Li and Shatnawi (2007) investigated the relationship between six code smells and class error probability in an industrial-strength system and found 'Shotgun Surgery' was positively associated with software faults. Monden et al. (2002) performed an analysis of a Cobol legacy system and concluded that cloned modules were more reliable, but required more effort than non-cloned modules. Rahman et al. (2010) conducted a descriptive analysis and non-parametric hypothesis testing of source code and bug tracker in four systems. They found that the majority of defective code was not significantly associated with clones (80% of defective code at system level contained zero clones); clones may be less defect-prone than non-cloned code and those that repeat less across the system are more error-prone than more repetitive clones.

Abbes et al. (2011) conducted an experiment in which twenty-four students and professionals were asked questions about the code in six open-source systems. They concluded that 'God Classes' and 'God Methods' alone had no effect, but code with the combination of 'God Class' and 'God Method' required a statistically significant increase in effort and decrease in correctness when compared with code without these smells. Deligiannis et al. (2003) conducted an observational study where four participants evaluated two systems, one compliant and one non-compliant with the principle of avoiding 'God Class'. Their main conclusion was that familiarity with the application domain played an important role when judging the negative effects of god class. They also conducted a controlled experiment (2004) with twenty-two undergraduate students as participants and could corroborate their initial findings that a design without a 'God Class' resulted in better completeness, correctness and consistency than that of the design that did contain a 'God Class'. Lozano and Wermelinger (2008) reported that existence of duplicated code increases maintenance effort on cloned methods. However, they were unable to identify characteristics revealing a significant relation between cloning and maintenance effort increase.

Khomh et al. (2009) analyzed the source code of Eclipse IDE and found that, in general, classes containing the 'Data Class' code smell was changed more often than classes without that smell. Kim et al. (2005) reported on the analysis of two medium-sized open source libraries (Carol and dnsjava) and concluded that of the 'Duplicated Code' smells, only 36% needed to be changed consistently; while the remainder of the duplications evolved independently. Olbrich et al. (2010) reported an experiment involving the analysis of three open-source systems and found that 'God Class' and 'Brain Class' smells were changed less frequently and had fewer defects than other classes when normalized with respect to size.

None of the aforementioned studies have explored the use of code smells for determining the maintainability of a system, nor evaluated its accuracy and descriptive richness in comparison to established methods for maintainability assessments. Also, none have compared their results across several systems, which constrains their internal and external validities. An additional knowledge gap is that these studies consider effects of individual code smells on individual maintenance measures located at class/method level. The applicability of their results for maintainability assessments is still very limited, because currently there is no knowledge on severity levels of each of the twenty-two code smells suggested by Fowler; there is also no guidance on how to aggregate them at the system level and interpret those aggregations to perform system-level assessments of maintainability.

Examples of automated smell detection work can be found in (Marinescu and Ratiu, 2004; Marinescu, 2005; Moha et al., 2006; Moha, 2007; Moha et al., 2008; Rao and Reddy, 2008; Alikacem and Sahraoui, 2009; Khomh et al., 2009; Moha et al., 2010). This has lead to commercial tools such as Borland Together (Borland, 2012) and InCode (Intooitus, 2012)as well as academic tools such as JDeodorant (Fokaefs et al., 2007; Tsantalis et al., 2008) and iSPARQL (Kiefer et al., 2007).

2.3. Maintainability Evaluation

Maintainability is one of the software qualities defined by (ISO/IEC, 1991) as: "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment and in requirements and functional specifications". To evaluate maintainability, numerous conceptual surrogates have been defined alongside software quality and measurement frameworks. Recent work on maintainability models can be found in (Kajko-Mattsso et al., 2006; Pizka and Deissenboeck, 2007) as well as in technical standards defined by ISO (ISO/IEC, 2005).Oman and Hagemeister (1992) suggests a taxonomy for maintainability measures and Kitchenham et al. (1999) define an ontology for software maintainability as an attempt to identify and describe the major maintenance factors influencing software maintenance processes.

In software engineering research, relatively high emphasis has been given to product and process factors for estimating maintenance effort and for maintainability assessments. Many product-centered approaches focus on generating prediction models by using historical data, using or refining previously defined software measures, or suggesting methodologies for selecting, structuring and interpreting software measures. Product-centered approaches that use software measures for evaluating maintainability can be found in (Oman and Hagemeister, 1994; Ferneley, 1999; Mathias et al., 1999; Muthanna et al., 2000; Fioravanti, 2001; Welker, 2001; Alshayeb and Wei, 2003; Succi et al., 2003; Benestad et al., 2006; Heitlager et al., 2007). A literature review of maintenance cost estimation models can be found in (Koskinen and Tilus, 2003), and a systematic literature review on the topic of maintainability evaluation and metrics can be found in (Riaz et al., 2009). Some views have commented on traditional OO metrics' lack of 'control' and difficulty in interpretation (Marinescu, 2002; Alshayeb and Wei, 2003; Heitlager et al., 2007). Benestad et al. (2006) assert that the biggest challenges of metrics-based assessment of system quality is the selection of adequate metrics, achieving a combination of metrics at system level that makes sense and, ultimately, their interpretation. Other approaches suggest combining code analysis with other approaches. One example is that by Mayrand and Coallier (1996), who combine capability assessment (based on the ISO/IEC-12207 standard) and code analysis. Work by Rosqvist et al. (2003) suggests combining code analysis with expert judgment.

3. Case study

3.1. Systems Studied

To conduct a longitudinal study of software development, the Software Engineering Department at Simula Research Laboratory put out a tender for the development of a new web-based information system to keep track of their empirical studies. Based on the bids, four Norwegian consultancy companies were hired to independently develop a version of the system, all using the same requirements specification. More details on the initial project can be found in (Anda et al., 2009). The systems developed were thus functionally equivalent, and constitute the same systems studied by Anda in (Anda, 2007).

The same four functionally equivalent systems are used in the study presented. We henceforward refer to them as System A, B, C and D, respectively. The systems were primarily developed in Java, had similar three-layered architectures, but had considerable differences in their design and implementation.

Their cost also differed notably (See Table 1), as the companies that were hired also differed notably in their bids. The bid price may have been affected by business factors within the companies (*e.g.,* different business strategies to profit from a project; some companies were willing to bid low to enter a new market). As reported in (Anda et al., 2009), the choice of the four companies was a trade-off between having a sufficient number of observations (projects) and having sufficient means to hire the companies and observe their projects.

Table 1: Development costs for each system Anda et al. (2009)

System	**A**	**B**	**C**	**D**
Cost	€25,370	€51,860	€18,020	€61,070

The main functionality of the systems consisted of keeping a record of the empirical studies and related information at Simula (*e.g.,* the researcher responsible of the study, participants, data collected and publications resulting from the study). Another key element of functionality was to generate a graphical report on the types of studies conducted per year. The systems were all deployed over Simula

Research Laboratories Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to a database in the CMS to access data related to researchers at Simula as well as information on the publications therein.

3.2. Previous Evaluations

In (Anda, 2007), two different approaches were used to evaluate and compare the maintainability of the same aforementioned systems when they went fully operational. One was based on software measures adapted from the C&K metrics (Chidamber and Kemerer, 1994) and a second approach based on expert judgment. The first approach draws heavily on the work by Benestad et al. (2006), who also used the same systems for their analysis.

The structural properties of the four systems were measured through an adapted version of the C&K metrics based on guidelines from (Bieman and Kang, 1995) and via principal component analysis (PCA) (Benestad et al., 2006). PCA was used to ensure that all measures were orthogonal implying that the same code factors would not be measured more than once. The list and description of the set of measures (also used in (Anda, 2007)) is presented in Table 2. The measurements were extracted via the M-System from Fraunhofer IESE (Ochs, 1998).

Table 2: List of software measures used in (Anda et al., 2009)

Structural measure	**Description**
Lines of code (LOC)	Number of non-commented lines of code
Comments	Number of comments in the code
Classes	Number of classes in the system
Weighted methods per class (WMC)	Number of methods in a class
Calls to methods in unrelated class (OMMIC)	Number of calls that a class has to unrelated classes
Calls from methods in unrelated class (OMMEC)	Number of calls that a class has to unrelated classes
Number of children (NOC)	Number of classes that inherit directly from the current class
Depth of inheritance tree (DIT)	Number of classes that are parents of a class, with the top-most class having a count of one.
Tight class cohesion (TCC)	Ratio of the number of method pairs of directly connected public methods in a class and the number of maximal possible method pairs of connections between the public methods of a class.

Subsequently, two different approaches were used to aggregate these measures at system level to evaluate the maintainability of the systems: *aggregation first* and *combination first* (these terms are taken from (Benestad et al., 2006)). In the first approach, all the measures were aggregated into summary statistics for the four systems (Table 3).

Table 3: Measures with *aggregation-first* approach in as described in (Anda, 2007)

System	**A**	**B**	**C**	**D**
LOC	7937	14549	7208	8293
Comments	1484	9135	1412	2508
Classes	63	162	24	96
WMC	6.9/11.2	7.8/10.3	11.4/12.5	4.9/4.5
OMMIC	7.7/15.8	5.3/11.8	8.6/25	4.7/14.1
OMMEC	7.7/20.6	5.3/15.6	8.6/16	4.7/10.1
NOC	0.46/2.75	0.59/2.37	0/0	0.76/3.81
DIT	0.46/0.5	0.75/0.81	0/0	0.83/0.54
TCC	0.26/0.37	0.17/0.31	0.20/0.23	0.11/0.22

In the second approach, the different measures were combined first per class by using a technique called *profile comparison* (Morisio et al., 2002). This technique consisted of using *profile vectors*, which define threshold values for each measure in order to categorize artifacts into groups.

For example, given three supposed measures: m_1, m_2 and m_3, for a given class to be categorized as "Very high", the measurement values for those measures should start at 20, 30 and 100, respectively. Consequently, the profile vector would be: *Very High* $= \{m_1 >= 20, m_2 >= 30, m_3 >= 100\}$. The measurements per class are iteratively compared to each profile vector, testing whether a "sufficient" majority of measures support the categorization and if is not "strongly" opposed by a minority of observations. For example, given our previous profile vector, if a class A has $m_1 = 22, m_2 = 20$ and $m_3 = 150$, it would be categorized as "Very high", given a rule that: "majority of measures support the classification".

In the work reported in (Benestad et al., 2006), four categories were used: Low, Average, High and Very High. The threshold values (profile vectors) and weights for the measures can be found in (Benestad et al., 2006). Each class was then classified into these four categories, according to how large their measures were. The limits of each category were calculated from 0-50 percentile, 50-75, 75-90 and above 90 percentile of all classes. The rule used to categorize a class was: "The weighted sum of the criteria supporting the categorization should be larger than the weighted sum opposing it".

Table 4 displays the structural measures for the combination-first approach. In this way, it is possible to evaluate for each system, the proportion of "potentially problematic" classes and which are expected to exhibit High or Very High values on their software measures. Based on the values from Table 3 and Table 4, Anda concluded that System C[2] had large and complex classes, uneven design, and

[2]In the work by Anda, the systems were also named A, B, C and D.

no use of inheritance. System D had a simple design and low coupling, despite relatively high use of inheritance. System A had many elements coupled with large dispersion values, but relatively low usage of inheritance. System B was deemed more maintainable than System A due to lower coupling measures.

Table 4: Measures with *combination-first* approach (Anda, 2007)

System	**A**	**B**	**C**	**D**
Low	41	87	7	58
Average	12	40	9	30
High	8	30	6	6
Very high	2	5	2	2

When observing the measures with combination-first, Anda observed that Systems A, C and D had few classes with high or very high values, although in System C, the low total number of classes made this percentage large. Finally, System B was difficult to judge since it displayed many classes with high/very high values, but also many with low/acceptable values. Anda's interpretation of structural measures resulted in System D being the most maintainable and System C the least maintainable. Systems A and B had very similar measurement values: in aggregation-first, B was deemed better than A due to lower values on coupling measures and in the combination-first, A was deemed better due to lower number of classes with "High"/ "Very High" values.

Table 5 presents the ranking resulting from the interpretation of the software measures, where: 1 = most maintainable and 4 = least maintainable. Anda remarks on the contingent nature of evaluations based on combined-first, given that it is difficult to foresee the impact of class size classes containing high values on the total maintainability of a system.

Table 5: Maintainability ranking by Anda using C&K metrics

System	**A**	**B**	**C**	**D**
Ranking	2	3	4	1

The second approach for maintainability evaluation by Anda was based on expert judgment. The decision of comparing software measures and expert judgment -based maintainability evaluations was inspired on the findings by Jø rgensen (2007). The study by Jørgensen reported that combining expert assessment and formal methods usually provided the best results for software effort estimations.
Anda analyzed a maintainability report on the systems written by two experienced software engineers, the first with more than twenty years of development experience, and the second expert with ten years of experience. A summary of the evaluation is given in Figure 1, which formed part of the rationale used by the experts to perform the ranking of the systems.

- System A is likely to be the most maintainable system, at least as long as the extensions to the system are not too large.
- System D exhibited slightly more potential maintainability problems than did System A, especially as some of the code was unfinished due to ambitions that were not fulfilled. However, System D may be a good choice if the system is to be extended significantly.
- System C was considered difficult to maintain. It may be easy to perform small maintenance tasks on the system, but it is not realistic to think that it could be extended significantly.
- System B was too complex and comprehensive and is likely to be very difficult to maintain. The design solution would have been more appropriate for a larger system.

Figure 1: Summary from expert's review on the maintainability of the systems

Table 6 displays the results from evaluation performed by both experts as part of Anda's study where: 1 = most maintainable and 4 = least maintainable. An important distinction made between the evaluations by Expert 1 and 2 was the emphasis given to system size. Expert 1 deemed system B to be the least maintainable while Expert 2 deemed C to be the least maintainable. According to Anda, the difference was due to "*...the fact that Expert 1 considered size and simplicity as more important for maintainability, while Expert 2 considered adherence to object-oriented principles as more important...*" (see Anda, 2007, pg.8). In that way, from a size perspective, B became worst due to its size and complexity, while C was favoured. From an OO design perspective, C was very "messy", while B had better structure.

Table 6: Maintainability ranking by Anda using expert judgment

System	**A**	**B**	**C**	**D**
Ranking Expert 1	1	4	3	2
Ranking Expert 2	1	3	4	2

3.3. Maintainability Evaluation Based on Code Smells

We used two commercial tools: Borland Together and InCode to detect code smells. This was done for several reasons: first, to facilitate repeatability of studies investigating code smells. Many studies report on findings based on smells detected by their own tools/methods and replication studies involving the same tools/methods are seldom reported. Several studies have used Borland Together to investigate the effects of code smells such as (Li and Shatnawi, 2007). Secondly, both tools are based on detection strategies (metrics-based interpretations of code smells) proposed by Marinescu (2002). We decided to measure as many types of code smells as possible to reduce false negatives.

Table 7 presents the list of code smells that were detected in the systems, alongside their descriptions, which were taken from (Fowler, 1999). It is important to note that Together can detect more code smells than those included in the table, but they were not included because we could not find any instances in the systems under analysis. Also, the last smell in the table is not a smell, but a design principle violation, as described by Martin (2002).

Table 7: Analyzed code smells and their descriptions, from (Fowler, 1999; Martin, 2002)

Smells	**Description**
Data Class	Classes with fields and getters and setters not implementing any specific function
Data Clumps	Clumps of data items that are always found together whether within classes or between classes
Duplicated code in conditional branches	Same or similar code structure repeated within a the branches of a conditional statement
Feature Envy	A method that seems more interested in another class other than the one it is actually in. Fowler recommends putting a method in the class that contains most of the data the method needs.
God (Large) Class	A class has the God Class smell if the class takes too many responsibilities relative to the classes with which it is coupled. The God Class centralizes the system functionality in one class, which contradicts the decomposition design principles.
God (Long) Method	A class has the God Method code smell if at least one of its methods is very large compared to the other methods in the same class. God Method centralizes the class functionality in one method
Misplaced Class	In "God Packages" it often happens that a class needs the classes from other packages more than those from its own package.
Refused Bequest	Subclasses do not want or need everything they inherit
Shotgun Surgery	A change in a class results in the need to make a lot of little changes in several classes
Temporary variable is used for several purposes	Consists of temporary variables that are used in different contexts, implying that they are not consistently used. They can lead to confusion and introduction of faults.
Use interface instead of implementation	Castings to implementation classes should be avoided and an interface should be defined and implemented instead.
Interface Segregation Principle Violation	The dependency of one class to another one should depend on the smallest possible interface. Even if there are objects that require non-cohesive interfaces, clients should see abstract base classes that are cohesive. Clients should not be forced to depend on methods they do not use, since this creates coupling

We used *aggregation-first* as the method to derive the code smell measures at system level, since it constitutes a simple and straightforward way to analyze code smells and no underlying assumptions are made. Another reason was that combination-first approach described in (Benestad et al., 2006) is based on profile vectors, and since currently there is no clear understanding of threshold values for code smells, aggregation-first was deemed most adequate for an exploratory study. In addition to plain smell aggregation, smell density was measured. Smell density is equal to the smells divided by size (LOC). This was done to adjust for size (Rosenberg, 1997). For the effects of our study, we calculated LOC as *physical LOC* (*i.e.,* including comment lines and blank lines), which were derived from SVNKit (TMate-Sofware, 2010).

3.4. The Maintenance Project

After the CMS of Simula Research Laboratory was replaced by a new platform called *Plone* (Plone Foundation, 2012), it was no longer possible to run the systems under this platform. Consequently, a maintenance project was required. This project was outsourced to two software companies in Eastern Europe at a total cost of 50.000 Euros.

Maintenance tasks. Three maintenance tasks were implemented as described in Table 8. Two tasks consisted of adapting the system to the new platform; a third task consisted of adding new functionality required by the users at that time.

Developers. Six developers from two software companies based in Eastern Europe individually conducted all three maintenance tasks. The developers were recruited from a pool of 65 participants in a study on programming skills (Bergersen and Gustafsson, 2011) that also included maintenance tasks. The developers were selected because their skill scores were $> 0\sigma$. They were also selected based on their availability, English proficiency and motivation for participating in the study.

Observations. Each of the six developers was asked to repeat the maintenance tasks on a second system, resulting in 3 observations (cases) per system. Thus, we make a distinction between "first round" cases and "second round" cases. "First round" denotes when a developer has not maintained any of the systems previously, and second round denotes the case when they are repeating the tasks on a second system.

Activities and Tools. The developers were given an overview of the project (e.g., the maintenance project goals, project activities) and a specification of each maintenance task. When needed, they would discuss the maintenance tasks with the researcher who was present at the site during the entire project duration. Daily meetings were held where progress and issues encountered were tracked. Acceptance tests were conducted once all tasks were completed and individual open interviews were conducted where the developer was asked his/her opinion of the system. MyEclipse (Genuitec, 2012) was used as the development tool, together with MySQL (Oracle, 2012) and Apache Tomcat (The Apache Software Foundation, 2012b). Defects were registered in Trac (Edgewall-Software, 2012)

Table 8: Maintenance tasks

No.	Task	Description
1	Adapting the system to the new Simula CMS	The systems in the past had to retrieve information through a direct connection to a relational database within Simula's domain (information on employees at Simula and publications). Now Simula uses a CMS based on Plone platform, which uses an OO database. In addition, the Simula CMS database previously had unique identifiers based on Integer type, for employees and publications, as now a String type is used instead. Task 1 consisted of modifying the data retrieval procedure by consuming a set of web services provided by the new Simula CMS in order to access data associated with employees and publications.
2	Authentication through web services	Under the previous CMS, authentication was done through a connection to a remote database and using authentication mechanisms available on that time for Simula Web site. This maintenance task consisted of replacing the existing authentication by calling a web service provided for this purpose.
3	Add new reporting functionality	This functionality provides options for configuring personalized reports, where the user can choose the type of information related to a study to be included in the report, define inclusion criteria based on people responsible for the study, sort the resulting studies according to the date that they were finalized, and group the results according to the type of study. The configuration must be stored in the systems' database and should only be editable by the owner of the report configuration.

(a system similar to Bugzilla), and Subversion or SVN (The Apache Software Foundation, 2012a) was used as the versioning system.

4. Maintainability Evaluation Based on Code Smells

Since the current state of art in code smell research has not yet investigated severity levels or differences in impact size amongst types of smells, it was not possible to calculate a weighted sum of the smells. Consequently, smells were simply added up to summarize them at system level. For smell density, number of smells were divided by *physical* LOC[3] and then added up at system level. The results are displayed in Table 9, where total smell, smell density and their standardized values are reported. The distribution of smells per type and Mean and SD are also reported.

Figure 2 shows, for each system, the standardized values for total number of smells and total smell density. If we consider total number of smells, System C has the lowest number of smells. Systems A and D have a similar number of smells, whereas System B displays the highest number of smells.

[3]Note that in the work by Anda, the definition of LOC is not physical. Nevertheless, the ratio between the systems remains the same for both interpretations of LOC.

Table 9: Measurement values and summary statistics of code smells

	System A		**System B**		**System C**		**System D**	
Physical LOC	8205		26662		5768		9942	
Total smells	111		161		44		97	
Total smell density	1.35283		0.60386		0.76283		0.97566	
Standardized value for smells	0.161		1.200		-1.231		-0.130	
Standardized value for smell density	1.324		-0.987		-0.497		0.160	
Code smell	*Qty.*	*Den.*	*Qty.*	*Den.*	*Qty.*	*Den.*	*Qty.*	*Den.*
Data class (DC)	12	0.146	32	0.120	9	0.156	24	0.241
Data clump (CL)	8	0.098	2	0.008	3	0.052	8	0.080
Duplicated code in conditional branches (DUP)	1	0.012	4	0.015	2	0.035	2	0.020
Feature envy (FE)	37	0.451	34	0.128	17	0.295	25	0.251
God class (GC)	1	0.012	5	0.019	3	0.052	2	0.020
God method (GM)	4	0.049	14	0.053	3	0.052	5	0.050
Interface Segregation Principle Violation (ISPV)	7	0.085	8	0.030	1	0.017	11	0.111
Misplaced class (MC)	0	0.000	2	0.008	0	0.000	2	0.020
Refused bequest (RB)	17	0.207	8	0.030	0	0.000	1	0.010
Shotgun surgery (SS)	7	0.085	17	0.064	0	0.000	13	0.131
Temporary variable used for several purposes (TMP)	12	0.146	31	0.116	6	0.104	4	0.040
Usage of implementation instead of interface (IMP)	5	0.061	4	0.015	0	0.000	0	0.000
Means: Smells = 103.25, Density = 0.92								
Standard Deviation: Smells = 48.11, Density = 0.32								

If we observe the LOC of each system (Table 9), we see that size is somewhat correlated to number of smells. An F test indicated a medium correlation (F ratio = 5.7822, p = 0.1380). Although this correlation is not statistically significant, it may hint that smells are influenced by the size of a system. This can be explained partially by the fact that many of the detection strategies are based on measures

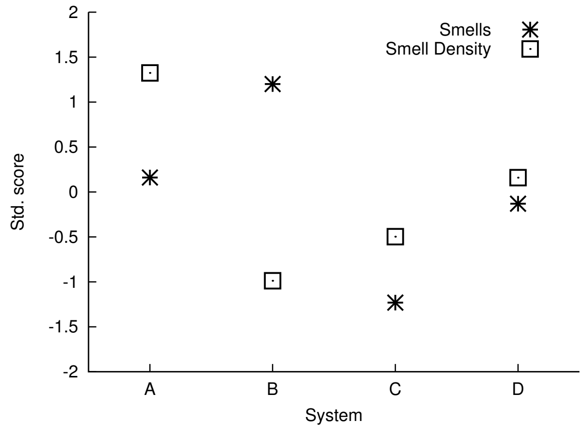


Figure 2: Standardized number of smells and density

related to size. For example, the God method is based on at least three size measures: LOC, Number of local variables (NOLV), and Maximum number of branches (MNOB). The fourth measure, Number of parameters (NOP) it could be argued is size-related. Figure 3 displays the detection strategy for God method (Marinescu, 2002), which can be translated to: from the top 10% methods in terms of LOC larger than 70 LOC, select those methods that have more than 4 NOP or more than 4 NOLV and the maximum number of if/else or case branches is bigger than 4.

```
GodMethod :=    (LOC, TopValues(%20)) butnotin (LOC, LowerThan(70)) and
                ((NOP, HigherThan(4) or (NOLV, HigherThan(4))) and
                (MNOB, HigherThan(4))
```

Figure 3: Detection strategy by Marinescu (2002) for God Method

We conducted one-way ANOVA and/or bivariate analysis (depending on whether a smell was a binary or continuous) to compare means between smells and LOC per class. Table 10 shows that the smells: Duplicated code in conditional branches, Feature Envy, God Class, God Method, ISP Violation, Shotgun Surgery and Temporary variable all display p values lower than .0001, indicating a significant positive correlation with LOC. As for the smells Data Class, Data Clump, Misplaced Class, Refused Bequest, and Implementation instead of interface were not significantly correlated to LOC, and some indicated a negative correlation (*e.g.,* Data class displayed a T Ratio of -1.25085). Within the context of this study, we found that a significant number of smells were actually dependent on size.

Table 10: Correlation tests between code smells and LOC per class

Code Smell	**F Ratio / T Ratio**	**Sig.**
Data Class	-1.25085	0.2118
Data Clump	-0.37268	0.7096
Duplicated code in conditional branches	76.2955	$< .0001*$
Feature Envy	307.6406	$< .0001*$
God Class	14.00279	$< .0001*$
God Method	209.6445	$< .0001*$
ISP Violation	7.291262	$< .0001*$
Misplaced Class	0.5962	0.4405
Refused Bequest	-0.57043	0.5687
Shotgun Surgery	3.024965	0.0027*
Temporary variable used for several purposes	105.5841	$< .0001*$
Implementation instead of interface	2.6344	0.1054

When code smells are not adjusted for size (*i.e.,* smell density is not used) System C was deemed to have the highest maintainability. This system contained 44 smells in total, corresponding to 1.231 times less number of smells than the average number of code smells amongst all four systems. Also, this type of aggregation suggests System B had very poor maintainability, since it contained 1.2 times more code smells than the average number of smells (Table 9). Conversely, when smell density is considered (which adjusts for the size of the systems), System B becomes 'more' maintainable than C, because this system is relatively large compared to the other systems (1.47 times larger than the average physical LOC, and 4.6 times larger than its smallest counterpart, System C). Consequently, its smell density resulted very low (0.987 times lower than the average density according to Table 9). It is interesting to note that there was no significant difference between the number of smells and smell density for System D (see Figure 2). This was in contrast to Systems A and B, both of which display significant differences between the two measures. System C displays a medium difference between the two measures.

Table 11 presents an ordered ranking summarizing the evaluation based on code smells; 1 represents the least value of table entry; for example, System C contained the least number of smells. Assuming that the less smells in the system, the more maintainable a system is, System C resulted as the most maintainable according to number of smells, and B resulted as the most maintainable according to smell density.

Table 11: Maintainability ranking according to code smells

System	**A**	**B**	**C**	**D**
Number of smells	3	4	1	2
Smell density	4	1	2	3

5. Project Outcomes

In Section 3.4, we provided details on the maintenance project and described how developers were assigned two systems to work with. As such, we made a distinction between "first round" (when the developer works for the first time) and "second round" (when the developer had completed the tasks in one system and is asked to repeat the tasks in a second system) cases. The design of the study consisted of assigning two systems to each developer. That resulted in a total of twelve projects (6 developers x 2 systems). When this design choice was made, we expected that the learning effect would not affect the maintenance outcomes between systems maintained during the "first round" and the "second round". Contrary to our initial assumptions, we found that the learning effect and code reuse during the second rounds highly influenced the effort. The difference in the mean effort between first round projects and second round projects was 40.45 hours. On average, developers spent 2.03 times more in the first round compared to the second round. Using round as the independent variable resulted on F ratio of 8.2252 with significance level of 0.0167. For this reason, only observations from the first round were considered. This resulted in 6 observations in total: two observations for System A, one for System B, two for System C and one for D.

Table 12: Maintenance outcomes: means and standardized scores

Sys	**Obs.**	**Effort**	**Mean**	**SD**	**Std.Score**	**Defects**	**Mean**	**SD**	**Std.Score**
A	2	57.45 109.71	83.58	36.95	-0.011	18 12	15	4.2	0.210
B	1	130.77	130.77	Na	1.416	21	21	Na	1.329
C	2	63.63 54	58.81	6.81	-0.761	12 7	9.5	3.5	-0.816
D	1	62.66	62.66	Na	-0.644	10	10	Na	-0.723

Table 12 displays for each system, the number of observations (projects), and their corresponding *effort* (person-hours) and *defects introduced* (weighted sum). It also displays the mean and standard deviation (for Systems A and C, which had two observations), and the standardized scores for each measure. System B had the highest standardized score for effort (1.416) and defects (1.329), which ranks this system as the least maintainable. System A had a high variation between its observations in relation to effort (SD = 36.95 hours), but not in relation to defects introduced (SD = 4.2). Despite this variation, there is a clear tendency for this system to be less maintainable than Systems C and D. Systems are summarized in Table 13.

Table 13: Maintainability ranking according to empirical results

System	**A**	**B**	**C**	**D**
Ranking	3	4	1	2

Figure 4 is a diagrammatic view of the maintenance outcome scores. The standardized scores for effort and defects follow a very similar trend, except for System A, which deviated to a small degree. Based on these observations, System C was the most maintainable system, followed closely by System D. System A resulted with an intermediate maintainability level; System B was deemed to be the least maintainable system.

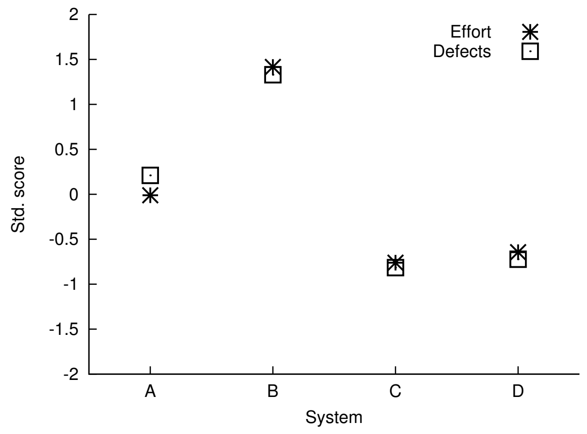


Figure 4: Maintenance scores for the systems

The data obtained from the open interviews supports the previous observations. Based on the interviews, general conclusions on the maintainability of the systems are the following:

- Systems A and C were considered rather small, and thus easy to understand and change (which may have provided them with certain advantage over systems B and D).
- Despite of its relatively small size, System A was fault-prone, and had confusing design (this might have reduced its advantage in relation to System D).
- System B was a big and complex system, which lacked flexibility and was hard to change.
- System C was the smallest, and thus easiest to perform changes on, but its high fault-proneness, lack of adequate technical platform, and having one big, unstructured class caused some delays in the project.
- System D had a medium size and a clear, understandable design.

Figure 5: Summary from open interviews

In the remainder of this sub-section, we will illustrate the points on each system with literal comments given by the developers during the interviews.

Comments on System A. Both developers who worked with the system reported that it was easy to get started with it due to its size (Comments 1, 2 in Figure 6). Nevertheless, it was deemed to be fault-prone (Comment 3). This is in line with the values for defects introduced. Potential reasons for this according to the developers was that system A contained design/implementation choices that "did not make sense", which eventually lead to confusion (Comment 4).

Comment 1: “A is a simple system and it was fast to start working with”

Comment 2: “A was not so difficult, since the application was relatively small”

Comment 3: “The biggest challenge was to make sure the changes made to the functionality wouldnt break the system, and the biggest difficulty was duplications in the code”

Comment 4: “In this system, data access objects were not only data access objects, they were doing a lot of other things”

Figure 6: Statements from developers on system A

Comments on System B. The developer who worked on System B mentioned that it was challenging to understand complex and non-standard frameworks and to understand the overall mechanics of the system in order to “get started” (Comment 1 in Figure 7). They also found System B’s mechanism for building queries to the DB difficult to learn and utilize (Comment 2). Modifications were considered time consuming and fault-prone due to the complex relationships of the classes involved (Comments 3 and 4). This complexity resulted in unmanageable ripple effects (Comment 5), which, in many cases, forced them to rollback and follow another strategy to solve the task. These statements can explain the high values on measures such as defects introduced.

Comment 1:“Spent long time learning how the system worked”

Comment 2:“The designer overdid it with a complicated framework with lots of layers and instruments. It is nonsense to use such a framework in current days”

Comment 3:“I spent long time extending it. The task was difficult because there were so many manual changes”

Comment 4:“It was more difficult in B to find the places to perform the changes because of the logic spread”

Comment 5:“There was no refactoring, that was considered too high risk”

Figure 7: Statements from developers on system B

Comments on System C. Developers who worked with System C agreed that this system had neither structure nor inheritance, although it was deemed as easy to learn (Comments 1, 2 in Figure 8). System C centralized the business logic into one large class, considered very fault-prone (Comments 3, 4). The arbitrary use of variables and the duplicated code were deemed as the main reasons for the introduction of faults in this class. This was deemed as the main reason for delays in the project (Comment 4).

Comment 1:"Understanding the system was easy, but it is maintainable because it is small"

Comment 2:"A messy system but not that complicated, so easy to learn"

Comment 3:"The big class in C was not easy to use, it could have been shorter"

Comment 4:"I had troubles with bugs in the big class, and mistakes in using different variables"

Figure 8: Statements from developers on system C

Comments on System D. In general, this system was deemed to have good design that was understandable and easy to work with (Figure 9). Despite system D being the second largest system, it scored as the second most maintainable system. It is likely that the good design in D gave an advantage over smaller systems such as A and C.

Comment 1:"System D was quite understandable"

Comment 2:"There were not many bugs"

Comment 3:"System D is balanced, no classes or methods that do too much"

Figure 9: Statements from developers on system D

6. Comparison of Maintainability Evaluation Approaches

To compare the different evaluation approaches, we decided to compare and analyze the degree of agreement between the maintainability rankings derived from each of the assessment approaches (Tables 5, 6 and 11). Also, to validate the approaches, we compared the degree of agreement between the rankings derived from the evaluations and the ranking resulting from the empirical results (Table 13). For example, if an evaluation approach 1 ranked the systems: A,D,C,B and the approach 2 ranked the systems B,C,D,A we would conclude that they have low agreement. Furthermore, if based on the empirical outcomes, it turned out that System A was the most maintainable and System B was the least maintainable, then we would conclude that approach 1 is the most accurate one. In order to derive a statistical measure for assessing the degree of agreement, Cohen's Kappa coefficient[4] was used.

[4]Cohen's Kappa coefficient is a statistical measure to represent inter-rater agreement for categorical items.

Table 14: Maintainability evaluations and maintenance outcomes

		High maint.	**Medium maint.**	**Low maint.**
Evaluations	Structural measures	D	A,B	C
	Expert judgment	A,D	C	B
	Number of code smells	C	D, A	B
	Code smell density	B	C,D	A
	Maintenance outcomes	C,D	A	B

Given that some evaluations ranked several systems similarly and the results from the maintenance project also showed at least one pair of systems displaying very similar results, we decided that a four-ranked ordinal scale would not give a fair comparison. Instead, we decided to group the systems into three categories: High maintainability, Medium maintainability and Low maintainability. Table 14 displays a merged version of the rankings reported in Tables 5, 6 and 11 adjusting the results into three categories instead of four.

Note that the assessment of the two experts as reported in Table 6 was merged into one to simplify the analysis. For example, the first row indicates that according to C&K based approach, System D was evaluated as the most maintainable, Systems A and B deemed as medium maintainable and System C as the least maintainable. The last row in the table corresponds to the ranking based on the actual maintenance outcomes (effort and defects), indicating that Systems C and D required least effort and least defects were introduced (thus, considered most maintainable) as System B required most effort and most defects were introduced (thus, the least maintainable).

The Kappa coefficient across all evaluations was rather low ($P_o = 0.20$). Table 16 provides a summary on the level of agreement between the different evaluation approaches. In (Anda, 2007), structural measures and expert assessment are to some extent aligned ($P_o = 0.25$, as shown in Table 15). If we divide the four systems into two groups instead: one with relatively good maintenance and one with relatively poor maintenance, structural measures and expert judgment mainly coincide, with A and D being acceptable systems and B and C being rather problematic. Numbers of smells agreed to an extent with previous evaluations, while smell density displayed the lowest level of agreement, pointing in many different directions ($P_o < 0.20$, thus not included in Table 15). If we re-calculate the Kappa excluding smell density, we obtain $P_o = 0.25$, which is the same level of agreement as structural measures and expert judgment.

When comparing the evaluation based on number of smells with structural measures, we see that they agree with respect to System A having an medium level of maintainability, but there is no matching for best or worst systems. When

Table 15: Comparison on levels of agreement between evaluations

Evaluation approaches	Level of matching or agreement	Kappa coefficient
Structural Measures vs. Expert Judgment	Matching in relation to A and D being acceptable systems and B and C being problematic.	0.25
Number smells vs. Structural Measures	Matching A as medium maintainable. No matching for the rest.	0.25
Number smells vs. Expert Judgment	Matching B as least maintainable. No matching for the rest.	0.25

compared to expert judgment, numbers of smells agree that System B is the least maintainable system. Finally, when comparing the evaluations with actual maintenance outcomes (See Table 16), number of code smells gave the best matching over previous evaluations (with $P_o = 0.75$): System C is most maintainable, A is medium maintainable and B is the least maintainable. System D was ranked with "medium maintainability" according to number of smells, but the empirical maintenance outcomes showed slightly better results. However, given that System D was still ranked after System C in the maintenance outcomes, this could also be seen as a relatively good match. In Table 16, the first row indicates the degree of matching of the evaluation based on C&K metrics compared to the maintenance project outcomes, and its corresponding Kappa coefficient. The evaluation coincides with the empirical ranking that D is most maintainable, and A is medium maintainable. Conversely, the last row indicates the degree of agreement of the evaluation based on smell density, which displayed zero matching with the maintenance outcomes.

Table 16: Comparison on levels of agreement with actual maintainability

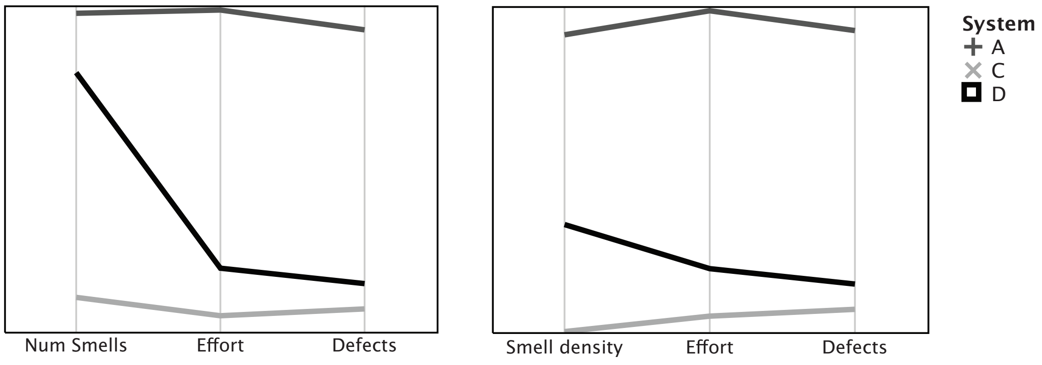
Evaluation approaches	Level of matching or agreement	Kappa coefficient
Structural Measures	Matching D as most maintainable and A as intermediate.	0.50
Expert Judgment	Matching D as most maintainable and B as least maintainable.	0.50
Number smells	Matching C as most maintainable, A as intermediate and B as least maintainable.	0.75
Smell density	No matching with maintenance outcomes	0.00

Smell density ranked System B as highly maintainable, providing a rather unrealistic comparison, at least for the size of maintenance tasks involved in this project. This result suggests that one should be careful when using smell density to compare systems that differ greatly in size, given that the effect of size on maintenance effort/defects could be masked by a density measure. However, if we remove System B from the analysis and only consider A, C and D, smell density can clearly discriminate the levels of maintainability given by the maintenance outcomes. Smell density ranked Systems C and D together, followed by System A, which is in accordance with the maintenance outcomes. In this scenario, smell density has a higher degree of agreement with the maintenance outcomes than number of smells.

Figure 10(a) shows a parallel plot of the standardized scores for: number of smells, effort and defects. The left side of the figure consists of the distance between Systems A, C and D in relation to the standardized scores on number of smells. The line for System C is drawn at the bottom, indicating the lowest number of smells, System D draws a line in the middle and System A is drawn at the top. In the middle of the figure, the standardized scores on effort are drawn. Finally, on the right side, standardized scores for defects are drawn.

Parallel plots are often used as visual aids when conducting *pattern matching* (Trochim, 1989). Pattern matching is the equivalent technique to hypothesis testing used in experimental settings, but applied to case study contexts. Lines in the plot are drawn between the corresponding theoretical expectations (or measures) and observed measures. Trochim (1989) asserts that the degree of correspondence (or match) between hypothesized measures and observed measures can be judged visually by the absence of crossed lines (crossovers) and the extent to which the lines remain parallel.

In Figure 10(a), there are no crossovers between the theoretical measures (i.e., standardized score distance across systems for number of smells) and the observed measures (i.e., standardized score distance across systems for effort and defects), but the degree of correspondence for System D is low, since for the number of smells, System D scored highly and this did not correspond to its results for effort and defects. Figure 10(b) displays the distance across systems on the standardized scores for smell density, effort and defects. We note that the distance between System C and System D is reduced, and consequently, the degree of correspondence between the evaluation measures and the maintenance outcomes improves considerably. Our preliminary results suggest that smell density tends to be sensitive to larger differences in system size, but would behave consistently if systems of similar size were compared. Given the assumption that the systems under analysis are of similar size, smell density can provide more informed results than just the sum of the number of smells per system.



(a) (b)

Figure 10: Parallel plots on the level of matching between the standardized scores of (a) *Number of smells* and (b) *Smell density*, versus maintenance outcomes Effort and Defects

7. Discussion

7.1. Findings

Results from this study indicate that counting the number of smells at system level can potentially be used for assessments of maintainability. In this study, the Kappa coefficient for number of smells displayed slightly higher accuracy than evaluations based on expert judgment and structural measures. It is important to note however, that this result applies to contexts where medium to small tasks constitute the typical maintenance scope. Large-scale maintainability projects would fall out of study scope. We found that seven out of the twelve smells measured in this study correlated strongly with class size. For some of the smells, this can be explained by the fact that their detection strategies are based on size measures. Also, since we sum up all the smells at system level, the final measure is also related to system size. Consequently, using plain aggregation of different code smells at system level may not provide any more information than LOC.

Smell density was used to adjust for size, but this measure conferred too much advantage to large systems. Because smell densities on extremely large systems become too small, this leads to unrealistic comparisons across systems highly dissimilar in size. However, when removing System B from the analysis, the ranking based on smell density matched better the maintenance outcomes than the ranking based on number of smells. Consequently, we can expect that when comparing systems with similar size, smell density can discriminate the system with the lowest maintainability. When comparing evaluation approaches, expert judgment was deemed the most practical of the approaches as it can be calibrated based on contextual information and can identify negative effects stemming from system's size and complexity. That was precisely the case for System B, which was judged by the first expert as the least maintainable, based on its size and complexity. The experts provided an alternative evaluation of System C:

> "It may be easy to perform small maintenance tasks on the system but it is not realistic to think that it could be extended significantly".

The maintainability level of System C is contingent on the size of the extension or change request to be implemented (a contextual factor). Evaluations incorporating different maintenance scenarios are more accurate than evaluations providing one universal perspective.

However, it seems as experts slightly misjudged System A, since it turned out to be less maintainable than predicted. The system displayed high numbers of code smells in comparison with C and D, and this was particularly visible when observing its smell density. Two developers worked in this system and they both agreed that its design was inconsistent and fault-prone.

These results stress the importance of incorporating contextual information (*e.g.,* size of maintenance tasks, maintenance scope). It is difficult to introduce context-related variables into an evaluation based on code measures whether structural measures or code smells; the most reasonable alternative would be to use expert assessment to interpret the measures. This has already been suggested by Anda (2007), and our results support the conclusions of that work. Given that code smells and LOC are somewhat related, one can still question the advantages of code smell -based evaluations over the use of more simple measures such as LOC. Code smells provide more information on the nature of the systems' design shortcomings than simply their size. Also, code smells can provide guidelines on how to reduce not only the size of the code, but how to improve aspects such as data-flow and functionality distribution to cope with a larger and more complex code base. Of course, in some circumstances, architectural changes may be needed to cope with system level issues and in such situations code smells may only provide limited support. Examples of architectural evaluation methods can be found in (Knodel et al., 2006; Folmer and Bosch, 2007), surveys on software architecture are reported in (Koziolek, 2011), and a set of architecture evaluation criteria is reported in (Bouwers et al., 2009).

7.2. Validity of Results

We consider the validity of the study presented from three perspectives:

Construct validity. We defined maintainability through two widely accepted measures: effort and number of defects introduced. They constitute a straightforward indication of maintainability. For each of these measures, we collected data from several data sources and this allowed us to triangulate[5] and ensure accuracy. With respect to code smells, we used automated detection to avoid subjective bias. Automated detection may still have false negatives, but the purpose of the study was also to evaluate the usefulness of the code smell definitions and their respective detection strategies to assess maintainability.

Internal validity. Several moderator factors were controlled for to ensure internal validity. For instance, the fact that all the evaluations used the same systems removes much of the variability that often comes with the system's context. The maintenance tasks were also controlled for, and special attention was given to ensure a similar environment and development technology across projects. Within the maintenance project, particular effort was spent on recruiting developers with as similar skills as possible, by using a skill instrument reported in peer-reviewed work (Bergersen and Gustafsson, 2011). For all developers with the exception of one, their skill scores were $> 0\sigma$. The skill scores of the instrument were derived from following the principles in (Bergersen et al., 2011), where performance on each task was scored as an structured aggregate of the quality (or correctness) and time for a correct solution for each task. Each task performance was subsequently scaled using the polytmous Rasch model (Andrich, 1978) and validated against test of working memory and Java programming knowledge reported in (Bergersen and Gustafsson, 2011).

External validity. The results from this study should be interpreted within the context of medium sized Java web information systems and medium to small maintenance tasks. The average effort for Tasks 1 and 2 were approximately 26 hours and for Task 2 it was approximately 6 hours. Despite the fact that in software engineering there is still no well-defined classifications of maintenance tasks (Sjøberg et al., 2007), we can assume that Tasks 1 and 3 constitute medium sized maintenance tasks and Task 2 constitutes a small maintenance task. It is possible to argue that they represent typical maintenance scenarios, given that they are based on real maintenance needs. Within software engineering corpus, a very small segment of studies incorporate in-vivo analysis, and in our case, we count with qualitative data to support the quantitative observations. We also consider the length of the total project to be non-trivial.

[5] In the social sciences, triangulation is often used to indicate that more than two methods are used in a study with a view to double (or triple) checking results. This is also called "cross examination"(Yin, 2002).

This study does not constitute a longitudinal study and, as such, its results are contingent on the context of an "acquisition project", where developers maintain the system for the first time. This study constitutes the first attempt to assess the usefulness of code smells for evaluating maintainability at system level. The rationale on how to derive system-level measures was based on previous work involving structural measures.

8. Conclusions and Future Work

In this study, we evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment and the C&K suite of metrics. The results from all three evaluations were compared against empirical data resulting from a maintenance project where several change requests were implemented in the systems.

We found that most of the code smells detection strategies used were based on size-related measures and given that we sum up the smells at system level, total measures were correlated to system size. A consequence of this is that when comparing systems varying in size, code smells may not provide any more information than that provided by LOC. Smell density was used to adjust for size, but this measure conferred too much advantage on large systems. Because smell densities on extremely large systems become too small, this leads to unrealistic comparisons across systems highly dissimilar in size. However, when comparing systems with similar size, smell density discriminated the system with the lowest maintainability according to the empirical maintenance outcomes. Consequently, code smells may not be very useful for comparing systems that differ considerably in size, but can potentially be useful for comparing systems of similar size.

When comparing code smells with other evaluation approaches, we found that structural measures provided more insight into which system had the most "balanced design", but this measure ignored the effect of size when maintenance tasks were of small/medium size. Expert judgment was found to be the most versatile of all three approaches, since it considered both the effect of system size and potential maintenance scenarios (*e.g.,* small extensions vs. large extensions). One advantage of code smells is that when comparing similar sized systems, they can spot critical areas that experts may overlook due to lack of time.

The maintenance outcomes are contingent on the nature and size of the tasks. Some tasks may influence certain areas of code that may or may not contain smells, and this is not reflected in the system-level analysis. Further work will attempt to analyze how well code smells can reflect potential maintenance problems, based on the analysis of class-level effort and maintenance issues/difficulties reported on areas of the code that were actually inspected and/or modified by the developers during the maintenance project.

Acknowledgements. The authors thank Gunnar Bergersen for support in selecting the developers of this study and Hans Christian Benestad for providing technical support in the planning stage of the study. Also, our thanks to Bente Anda, Dag Sjøberg and Magne Jørgensen for useful discussions that lead to the paper. This work was partly funded by Simula Research Laboratory, and the Research Council of Norway through the projects AGILE, grant no. 179851/I40, and TeamIT, grant no. 193236/I40.

Authors' Bio:

Aiko Yamashita received the BSc. degree from Costa Rica Institute of Technology in 2004 and the MSc. degree from Göteborgs University in 2007. She is currently finalizing her doctoral degree at Simula Research Laboratory and The Department of Informatics, University of Oslo. She has worked five years as a software engineer and consultant in Costa Rica, USA, Sweden and Norway within diverse organizations. Her research interests include empirical software engineering, software quality, psychology of programming, HCI and agile methods.

Steve Counsell is a Reader in the Department of Information Systems and Computing at Brunel University. He received his PhD from Birkbeck, University of London in 2002 and his research interests relate to empirical software engineering; in particular, refactoring, software metrics and the study of software evolution. He worked as an industrial developer before his PhD.

References

Abbes, M., Khomh, F., Gueheneuc, Y.G., Antoniol, G., 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension, in: European Conf. Softw. Maint. and Reeng., pp. 181–190.

Abran, A., Nguyenkim, H., 1991. Analysis of maintenance work categories through measurement, in: Int'l Conf. Softw. Maint., pp. 104–113.

Alikacem, E.H., Sahraoui, H.A., 2009. A Metric Extraction Framework Based on a High-Level Description Language, in: Working Conf. Source Code Analysis and Manipulation, pp. 159–167.

Alshayeb, M., Wei, L., 2003. An empirical validation of object-oriented metrics in two different iterative software processes. IEEE Trans. Softw. Eng. 29, 1043–1049.

Anda, B., 2007. Assessing Software System Maintainability using Structural Measures and Expert Assessments, in: Int'l Conf. Softw. Maint., pp. 204–213.

Anda, B.C.D., Sjøberg, D.I.K., Mockus, A., 2009. Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System. IEEE Trans. Softw. Eng. 35, 407–429.

Andrich, D., 1978. A rating formulation for ordered response categories. Psychometrika 43, 561–573.

Benestad, H., Anda, B., Arisholm, E., 2006. Assessing Software Product Maintainability Based on Class-Level Structural Measures, in: Product-Focused Softw. Process Improvement, pp. 94–111.

Bennett, K.H., 1990. An introduction to software maintenance. Inf. and Softw. Tech. 12, 257–264.

Bergersen, G.R., Gustafsson, J.E., 2011. Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective. J. of Individual Differences 32, 201–209.

Bergersen, G.R., Hannay, J.E., Sjøberg, D.I.K., Dybå, T., Karahasanovic, A., 2011. Inferring Skill from Tests of Programming Performance: Combining Time and Quality, in: Intl Conf. Softw. Eng. and Measurement, pp. 305–314.

Bieman, J.M., Kang, B.K., 1995. Cohesion and reuse in an object-oriented system, in: Symposium on Softw. Reusability, pp. 259–262.

Borland, 2012. Borland Together, url:www.borland.com/us/products/together.

Bouwers, E., Visser, J., van Deursen, A., 2009. Criteria for the evaluation of implemented architectures, in: Int'l Conf. Softw. Maint., IEEE. pp. 73–82.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493.

D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the Impact of Design Flaws on Software Defects, in: Int'l Conf. Quality Softw., pp. 23–31.

Deligiannis, I., Shepperd, M., Roumeliotis, M., Stamelos, I., 2003. An empirical investigation of an object-oriented design heuristic for maintainability. J. Syst. Softw. 65, 127–139.

Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M., Shepperd, M., 2004. A controlled experiment investigation of an object-oriented design heuristic for maintainability. J. Syst. Softw. 72, 129–143.

Edgewall-Software, 2012. Trac [online] Available at: http://trac.edgewall.org [Accessed 10 May 2012].

van Emden, E., Moonen, L., 2001. Java quality assurance by detecting code smells, in: Working Conf. Reverse Eng., pp. 97–106.

Ferneley, E.H., 1999. Design metrics as an aid to software maintenance: an empirical study. J. Softw. Maint. 11, 55–72.

Fioravanti, F., 2001. Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems. IEEE Trans. Softw. Eng. 27, 1062–1084.

Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A., 2007. JDeodorant: Identification and removal of feature envy bad smells, in: Int'l Conf. Softw. Maint., pp. 519–520.

Folmer, E., Bosch, J., 2007. A pattern framework for software quality assessment and tradeoff analysis. Int'l J. of Softw. Eng. and Knowledge Eng. 17, 515–538.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Genuitec, 2012. My Eclipse [online] Available at: http://www.myeclipseide.com [Accessed 10 May 2012].

Harrison, W., Cook, C., 1990. Insights on improving the maintenance process through software measurement, in: Int'l Conf. Softw. Maint., pp. 37–45.

Heitlager, I., Kuipers, T., Visser, J., 2007. A Practical Model for Measuring Maintainability, in: Int'l Conf. Quality of Information and Comm. Techn., pp. 30–39.

Intooitus, 2012. InCode [online] Available at: http://www.intooitus.com/inCode.html [Accessed 10 May 2012].

ISO/IEC, 1991. International Standard ISO/IEC 9126, International Organization for Standardization.

ISO/IEC, 2005. ISO/IEC Technical Report 19759:2005.

Jø rgensen, M., 2007. Estimation of Software Development Work Effort:Evidence on Expert Judgment and Formal Models. Int'l J. of Forecasting 23, 449–462.

Jones, T.C., 1998. Estimating software costs. McGraw-Hill.

Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S., 2009. Do code clones matter?, in: Int'l Conf. Softw. Eng., pp. 485–495.

Kajko-Mattsso, M., Canfora, G., van Deursen, A., Ihme, T., Lehman, M.M., Reiger, R., Engel, T., Wernke, J., 2006. A Model of Maintainability - Suggestion for Future Research, in: Reza, H.R.A., Hassan (Eds.), Software Engineering Research and Practice. CSREA Press, pp. 436–441.

Khomh, F., Di Penta, M., Gueheneuc, Y.G., 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness, in: Working Conf. Reverse Eng., pp. 75–84.

Kiefer, C., Bernstein, A., Tappolet, J., 2007. Mining Software Repositories with iSPAROL and a Software Evolution Ontology, in: Int'l Workshop on Mining Software Repositories, p. 10.

Kim, M., Sazawal, V., Notkin, D., Murphy, G.C., 2005. An empirical study of code clone genealogies, in: European Softw. Eng. Conf. (ESEC) and ACM SIGSOFT Symposium on Foundations of Softw. Eng. (FSE-13), pp. 187–196.

Kitchenham, B.A., Travassos, G.H., von Mayrhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., Yang, H., von Mayrhauser, A., 1999. Towards an ontology of software maintenance. J. Softw. Maint. 11, 365–389.

Knodel, J., Lindvall, M., Muthig, D., Naab, M., 2006. Static evaluation of software architectures, in: European Conf. Softw. Maint. and Reengineering, pp. 279–294.

Koskinen, J., Tilus, T., 2003. Software maintenance cost estimation and modernization support. Technical Report. Information Technology Research Institute, University of Jyvaskyla.

Koziolek, H., 2011. Sustainability evaluation of software architectures, in: ACM Sigsoft Int'l Conf. on the Quality of Softw. Architectures, pp. 3–12.

Lanza, M., Marinescu, R., 2005. Object-Oriented Metrics in Practice. Springer.

Li, W., Shatnawi, R., 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80, 1120–1128.

Lozano, A., Wermelinger, M., 2008. Assessing the effect of clones on changeability, in: Int'l Conf. Softw. Maint., pp. 227–236.

Mäntylä, M., 2005. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement, in: Int'l Conf. Softw. Eng., pp. 277–286.

Mäntylä, M., Vanhanen, J., Lassenius, C., 2003. A taxonomy and an initial empirical study of bad smells in code, in: Int'l Conf. Softw. Maint., pp. 381–384.

Mäntylä, M., Vanhanen, J., Lassenius, C., 2004. Bad smells -humans as code critics, in: Int'l Conf. Softw. Maint., pp. 399–408.

Mäntylä, M.V., Lassenius, C., 2006. Subjective evaluation of software evolvability using code smells: An empirical study. Empirical Software Engineering 11, 395–431.

Marinescu, R., 2002. Measurement and Quality in Object Oriented Design. Doctoral thesis. "Politehnica" University of Timisoara.

Marinescu, R., 2005. Measurement and quality in object-oriented design, in: Int'l Conf. Softw. Maint., pp. 701–704.

Marinescu, R., Ratiu, D., 2004. Quantifying the quality of object-oriented design: the factor-strategy model, in: Working Conf. Reverse Eng., pp. 192–201.

Martin, R.C., 2002. Agile Software Development, Principles, Patterns and Practice. Prentice Hall.

Mathias, K.S., Cross, J.H., Hendrix, T.D., Barowski, L.A., 1999. The role of software measures and metrics in studies of program comprehension, in: ACM Southeast regional Conf., p. 13.

Mayrand, J., Coallier, F., 1996. System acquisition based on software product assessment, in: Intl Conf. Softw. Eng., pp. 210–219.

Moha, N., 2007. Detection and correction of design defects in object-oriented designs, in: ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications, pp. 949–950.

Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F., 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Trans. Softw. Eng. 36, 20–36.

Moha, N., Gueheneuc, Y.G., Le Meur, A.F., Duchien, L., 2008. A domain analysis to specify design defects and generate detection algorithms. Fundamental Approaches to Softw. Eng. 4961, 276–291.

Moha, N., Gueheneuc, Y.G., Leduc, P., 2006. Automatic generation of detection algorithms for design defects, in: IEEE/ACM Int'l Conf. on Automated Softw. Eng., pp. 297–300.

Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K., 2002. Software quality analysis by code clones in industrial legacy software, in: IEEE Symposium on Software Metrics, pp. 87–94.

Morisio, M., Stamelos, I., Tsoukias, A., 2002. A new method to evaluate software artifacts against predefined profiles, in: Int'l Conf. on Softw. Eng. and Knowledge Eng., pp. 811–818.

Muthanna, S., Kontogiannis, K., Ponnambalam, K., Stacey, B., 2000. A maintainability model for industrial software systems using design level metrics, in: Working Conf. Reverse Eng., pp. 248–256.

Ochs, M., 1998. M-System - Calculating Software Metrics from C++ Source Code (Report no. 005/98). Technical Report. Fraunhofer IESE.

Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I., 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems, in: Int'l Conf. Softw. Maint., pp. 1–10.

Oman, P., Hagemeister, J., 1992. Metrics for assessing a software system's maintainability, in: Int'l Conf. Softw. Maint., pp. 337–344.

Oman, P., Hagemeister, J., 1994. Construction and testing of polynomials predicting software maintainability. J. Syst. Softw. 24, 251–266.

Oracle, 2012. My Sql [online] Available at: http://www.mysql.com [Accessed 10 May 2012].

Pigoski, T.M., 1996. Practical Software Maintenance: Best Practices for Managing Your Software Investment. Wiley.

Pizka, M., Deissenboeck, F., 2007. How to effectively define and measure maintainability, in: Softw. Measurement European Forum.

Plone Foundation, 2012. Plone CMS: Open Source Content Management [online] Available at: http://plone.org [Accessed 10 May 2012].

Rahman, F., Bird, C., Devanbu, P., 2010. Clones: What is that smell?, in: Intl Conf. Softw. Eng., pp. 72–81.

Rao, A.A., Reddy, K.N., 2008. Detecting bad smells in object oriented design using design change propagation probability matrix, in: Int'l Multiconference of Engineers and Computer Scientists, pp. 1001–1007.

Riaz, M., Mendes, E., Tempero, E., 2009. A systematic review of software maintainability prediction and metrics, in: Int'l Conf. Softw. Eng., pp. 367–377.

Rosenberg, J., 1997. Some Misconceptions About Lines of Code, in: Int'l Symposium on Softw. Metrics, pp. 137–142.

Rosqvist, T., Koskela, M., Harju, H., 2003. Software Quality Evaluation Based on Expert Judgement. Softw. Quality Control 11, 39–55.

Shanteau, J., 1992. Competence in experts: The role of task characteristics. Organizational Behavior and Human Decision Processes 53, 252–266.

Sjøberg, D.I.K., Dybå, T., Jø rgensen, M., 2007. The Future of Empirical Methods in Software Engineering Research. Future of Software Engineering (FOSE) SE-13, 358–378.

Succi, G., Pedrycz, W., Stefanovic, M., Miller, J., 2003. Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics. J. Syst. Softw. 65, 1–12.

The Apache Software Foundation, 2012a. Apache Subversion [online] Available at: http://subversion.apache.org [Accessed 10 May 2012].

The Apache Software Foundation, 2012b. Apache Tomcat [online] Available at: http://tomcat.apache.org [Accessed 10 May 2012].

TMate-Sofware, 2010. SVNKit - Subversioning for Java. [online] Available at: http://svnkit.com [Accessed 10 May 2012].

Travassos, G., Shull, F., Fredericks, M., Basili, V.R., 1999. Detecting defects in object-oriented designs, in: ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications, pp. 47–56.

Trochim, W., 1989. An introduction to concept mapping for planning and evaluation. Evaluation and program planning 12, 1–16.

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2008. JDeodorant: Identification and removal of type-checking bad smells, in: European Conf. Softw. Maint. and Reeng., pp. 329–331.

Wake, W.C., 2003. Refactoring Workbook. Addison-Wesley.

Welker, K.D., 2001. Software Maintainability Index Revisited. CrossTalk - J. of Defense Softw. Eng .

Yin, R., 2002. Case Study Research : Design and Methods (Applied Social Research Methods). SAGE.