Do code smells reflect important maintainability aspects?

Aiko Yamashita
Simula Research Laboratory &
Dept. of Informatics, University of Oslo, Norway
Email: aiko@simula.no

Leon Moonen
Simula Research Laboratory
Lysaker, Norway
Email: leon.moonen@computer.org

***Abstract*—Code smells are manifestations of design flaws that can degrade code maintainability. As such, the existence of code smells seems an ideal indicator for maintainability assessments. However, to achieve comprehensive and accurate evaluations based on code smells, we need to know how well they reflect factors affecting maintainability. After identifying which maintainability factors are reflected by code smells and which not, we can use complementary means to assess the factors that are not addressed by smells. This paper reports on an empirical study that investigates the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers. We consider two sources for our analysis: (1) expert-based maintainability assessments of four Java systems before they entered a maintenance project, and (2) observations and interviews with professional developers who maintained these systems during 14 working days and implemented a number of change requests.**

***Keywords*-maintainability evaluation; code smells;**

I. Introduction

Developing strategies for assessing the maintainability of a system is of vital importance, given that significant effort and cost in software projects is due to maintenance [1, 2]. Recently, existence of code smells has been suggested as an approach to evaluate maintainability [3]. Code smells indicate that there are issues with code quality, such as understandability and changeability, which can lead to the introduction of faults [4]. Beck and Fowler informally describe twenty-two smells and associate them with refactoring strategies to improve the design. Consequently, code smell analysis opens up the possibility for integrating both assessment and improvement in the software maintenance process.

Nevertheless, to achieve accurate maintainability evaluations based on code smells, we need to better understand the "scope" of these indicators, i.e. know their capacity and limitations to reflect software aspects considered important for maintainability. In that way, complementary means can be used to address the factors that are not reflected by code smells. Overall, this will help to achieve more comprehensive and accurate evaluations of maintainability.

Previous studies have investigated the relation between individual code smells and different maintenance characteristics such as effort, change size and defects; but no study has addressed the question of how well code smells can be used for general assessments of maintainability. Anda reports on a number of important maintainability aspects that were extracted from expert-judgement-based maintainability evaluations of four medium-sized Java web applications [5]. She concludes that software measures and expert judgment constitute not opposing, but complementary approaches because they both address different aspects of maintainability.

This paper investigates the extent to which aspects of maintainability that were identified as important by programmers are reflected by code smell definitions. Our analysis is based on an industrial case study where six professional software engineers were hired to maintain the same set of systems that were analyzed in [5]. They were asked to implement a number of change requests over the course of 14 working days. During this time, we conducted daily interviews and one larger wrap-up interview with each of the developers. We analyze the transcripts of these interviews using a technique called *cross-case synthesis* to compare each developer's perception on the maintainability of the systems and relate it back to code smells. The results from this analysis were compared to the data reported in [5].

The contributions of this paper are: (1) we complement the findings by Anda [5] by extracting maintainability factors that are important from the software maintainer's perspective; (2) based on manifestations of these factors in an industrial maintenance project, we identify which code smells (or alternative analysis methods) can assess them; and (3) we provide an overview of the capability of current smell definitions to evaluate the overall maintainability of a system.

The remainder of this paper is structured as follows: First we present the theoretical background and related work. Section 3 describes the case study and discusses the earlier results reported by Anda [5]. Section 4 presents and discusses the results from our analysis. Finally, Section 5 summarizes the study findings and presents plans for future work.

II. Theoretical background and related work

Maintainability assessments: There is a wealth of published research on product- and process based approaches for estimating maintenance effort and the related assessments of maintainability in software engineering literature. Examples of product-based approaches that use software metrics to assess maintainability include [6–8]. Some approaches use hierarchical quality models to relate external quality attributes (such as maintainability) to internal attributes and metrics.

This decomposition or Factor-Criteria-Metrics (FCM) approach was first used by McCall and Boehm [9]. Examples of process-centered approaches for maintenance effort estimation can be found in [10, 11]. Many of the process-centered approaches utilize process-related metrics or historical data to generate estimation models. Hybrid approaches combine process and product related factors: Mayrand and Coallier combine capability assessment (ISO/IEC-12207) with static analysis [12], and Rosqvist combines static analysis with expert judgment [13]. Riaz provides a systematic review of scientific literature on maintainability evaluation [14].

Factors affecting maintainability: Different code characteristics have been suggested to affect maintainability. Early examples include size (lines of code, LOC) and complexity measures by McCabe [15] and Halstead [16]. Some have attempted to combine them into a single value, called maintainability index [17]. Measures for inheritance, coupling and cohesion were suggested in order to cope with object-oriented program analysis [18].

Pizka & Deissenboeck [19] assert that, even though such metrics may correlate with effort or defects, they have limitations for assessing the overall maintainability of a system. One major limitation is that they only consider properties that can be automatically measured in code, whereas many essential quality issues, such as the usage of appropriate data structures and meaningful documentation, are semantic in nature and cannot be analyzed automatically. Anda [5] reported that software metrics and expert judgment are complementary approaches that address different maintainability factors. Important factors that are not addressed by metrics are: Choice of classes and names, Usage of components, Adequate architecture, Design suited to the problem domain.

Code smells: A code smell is a suboptimal design choice that can degrade different aspects of code quality such as understandability and changeability, and could lead to the introduction of faults [4]. Beck and Fowler [4] informally describe 22 code smells and associated them with refactoring strategies to improve the design. In the last decade, code smells have become an established concept for patterns or aspects of software design that may cause problems for further development and maintenance of the system [3]. Code smell analysis allows for integrating both assessment and improvement in the software evolution process. Moreover, code smells constitute software factors that are potentially easier to interpret than traditional OO software measures, since many of the descriptions of code smells in [4] are based on situations that developers face in a daily basis.

Van Emden and Moonen [20] provided the first formalization of code smells and developed an automated code smell detection tool for Java. Mäntylä [21] and Wake [22] proposed two initial taxonomies for code smells. Mäntylä investigated how developers identify and interpret code smells, and how this compares to results from automatic detection tools [23]. Examples of recent approaches for code smell detection can be found in [24–27]. Automated detection is implemented in commercial tools such as Borland Together[1] and InCode[2].

Previous empirical studies have investigated the effects of individual code smells on different maintainability related aspects, such as defects [28, 29], effort [30–32] and changes [26, 33]. One of the main goals of incorporating code smells to maintainability evaluations is to address a limitation that expert judgment and traditional code metrics have in common: for both approaches, there is no clear path from evaluation to concrete action plans for improvement. As Anda [5] points out, if one asks an expert to identify the areas to modify to improve maintainability, it would be time-consuming and expensive. Likewise, Marinescu [24] and Heitlager [8] point out that a major limitation of metrics is their lack of guidelines to improve their value (and thereby maintainability). Code smells do not suffer from these drawbacks due to their associated refactorings. Moreover, since an increasing number of code smells can be detected automatically, it is appealing to evaluate their capacity to uncover different factors that affect maintainability. The extent to which we understand how well code smells cover these factors determines our ability to address their limitations by alternative means. This will support more comprehensive and cost-effective evaluations of software maintainability.

III. Case Study

Systems under analysis: To conduct a longitudinal study of software development, the Simula's Software Engineering Department put out a tender in 2003 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. The companies knew, and agreed that the work would be done as part of a research study. More details on the initial project can be found in [34].

The same four functionally equivalent systems are used in our current study. We will refer to them as System A, System B, System C and System D, respectively. The systems were primarily developed in Java and have similar three-layered architectures, but have considerable differences in their design and implementation. The systems were deployed over Simula's Content Management System (CMS), which at that time was based on PHP and a relational database system. The systems had to connect to the database in the CMS, in order to access data related to researchers at Simula as well as information on the publications.

Software factors important to maintainability: After the systems were developed, two (external) professional software engineers were hired to individually evaluate the maintainability of these systems. The first software engineer had more

[1] http://www.borland.com/us/products/together
[2] http://www.intooitus.com/products/incode

Table I
IMPORTANT FACTORS AFFECTING MAINTAINABILITY, AS REPORTED IN [5]

Factor	Description
Appropriate technical platform	Related to undocumented, implicit requirements surfacing when a system is moved to a different environment. The use of non-standard third party components poses a challenge to understanding, using and replacing the components in further development.
Coherent naming	Developers should use a consistent naming schema that allows the reader to understand relations between methods and classes. Classes should be easy to identify to facilitate the mapping from domain and requirements to code.
Comments	Comments should be meaningful and size-effective, so they do not influence negatively the readability of the code.
Design suited to problem domain	The complexity of the problem domain must justify the choice for the design. For example, the use of design patterns must be adapted to the project context.
Encapsulation	Since Java methods return only one object, developers often create small "output" container classes as a work-around. This introduces dependencies, such as creating object structures before a method is called, which can lead to maintenance problems.
Inheritance	The use of inheritance increases the total number of classes, so therefore should be used with care. If an interface implements several classes, it has the same effect as multiple inheritances, which may lead to confusion and lower maintainability.
Libraries	The use of proprietary libraries may mean lower maintainability, because new developers will need to familiarize themselves with them.
Simplicity	Size and complexity of a system is critical. It takes longer to identify a specific class when there are many classes. The presence of several classes that are almost empty is a sign of code that may possess low maintainability.
Standard naming conventions	The use of standard naming conventions for packages, classes, methods and variables eases understanding.
Three-layer architecture	A clear separation of concerns between presentation, business and persistence layer is considered good practice. Each layer should remain de-coupled from the layers above it and depend only on more general components in the lower layers.
Use of components	Classes should be organized according to functionality or according to the layer of the code on which they operate.

than 20 years of experience at that time, and the second expert had 10 years of experience. The following is an excerpt of their maintainability assessment based on expert judgment, sorted from highest- to lowest maintainability.

- System A is likely to be the most maintainable, as long as the extensions to the system are not too large.
- System D shows slightly more potential maintainability problems than System A. However, System D may be a good choice if the system is to be extended significantly.
- System C was considered difficult to maintain. Small maintenance tasks may be easy, but it is not realistic to think that it could be extended significantly.
- System B is too complex and comprehensive and is likely to be very difficult to maintain. The design would have been more appropriate for a large-scale system.

From the full evaluations, Anda extracted factors that affect maintainability, and concluded that most of them are only addressable by expert judgment, and not by metrics [5]. An overview of these factors is shown in Table I.

Maintenance project: In 2008, Simula's CMS was replaced by a new platform called Plone,[3] and it was no longer possible to run the systems under this new platform. This gave the opportunity to set up a maintenance study, where the functional similarity of the systems enabled investigating the relation between design aspects and maintainability on cases with very similar contexts (e.g., identical tasks and programming language), but different designs and implementations. This study was conducted between September and December 2008 by outsourcing the project to two software companies in Eastern Europe at a total cost of 50.000 Euros.

Maintenance Tasks: Three maintenance tasks were defined, as described in Table II. Two tasks concerned adapting the system to the new platform and a third task concerned the addition of new functionality that users had requested.

Developers: Six developers were recruited from a pool of 65 participants in a study on programming skill [35] that included maintenance tasks. They were selected based on their availability, English proficiency, and motivation for participating in a research project.

Activities and Tools: The developers were given an overview of the project and a specification of each maintenance task. When needed, they would discuss the maintenance tasks with the researcher (first author) who was present at the site during the entire project duration. Daily interviews were held where the progress and the issues encountered were tracked. Acceptance tests were conducted once all tasks were completed, and individual open interviews were conducted where the developer was asked upon his/her opinion of the system. The daily interviews and wrap-up interviews were recorded for further analysis. MyEclipse[4] was used as the development tool, together with MySQL[5] and Apache Tomcat[6]. Defects were registered in Trac[7] (a system similar to Bugzilla), and Subversion or SVN[8] was used as the versioning system.

Research methodology: The process to extract the maintainability aspects from the developer interviews followed a chain of evidence strategy as shown in Figure 1.

Observed cases – Each of the six developers individually conducted all three tasks on two systems. This was done

[3]http://www.plone.org
[4]http://www.genuitec.com
[5]http://www.oracle.com
[6]http://tomcat.apache.org
[7]http://trac.edgewall.org
[8]http://subversion.apache.org

Table II
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.

to collect more observations for different types of analysis, and gave us a total of 12 cases, 3 observations per system. The assignment of developers to systems was random, with control for equal representation, learning effects (i.e. every system at least once in first round and at least once in second round) and maximizing contrast between the two cases handled by each developer (based on the expert-judgments).

Data collection and summarization – After the developers had finished the maintenance tasks for a one system, individual open-ended interviews (approx. 60 minutes) were held, where the developer was asked to give his/her opinion of the system and underlying reasons for the opinion. The choice for open-ended interviews is based on the rationale that important maintainability aspects should emerge naturally from the interview, and not be influenced by the interviewer. To enable data-triangulation, the daily interviews were transcribed and analyzed to collect data to cross-examine findings from the open-ended interviews. The daily interviews (20-30 minutes) resulted from individual meetings (mostly in the morning) with each developer, to keep track of the progress, and to record any difficulties encountered during the project (ex. Dev: "It took me 3 hours to understand this method..."). All recorded interviews were transcribed and summarized using a tool called Transana.[9]

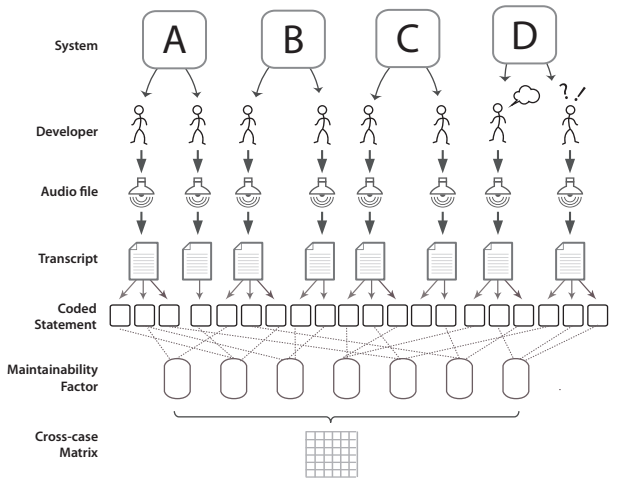

Figure 1. Chain of evidence for data summarization and analysis

Data analysis – The data was analysed using cross-case synthesis [36] and coding techniques [37]. Cross-case synthesis is a technique to summarize and identify tendencies in a multiple case study. Transcripts from the interviews were coded using both open and axial coding, as described in [37]. Open coding is a form of content analysis. Statements from the developers were annotated using labels (codes) that were initially constructed from a logbook that the on-site researcher kept during the project, and iteratively revised during the annotation process. Example codes are: DB Queries, Size, Bad naming, Lack of rules, Data Access Objects.

For axial coding, the annotated statements were grouped according to the most similar concepts, based on the researchers' observations throughout the project. For more details we refer to our technical report [38], Appendix A. During this process, we found that many of the categories were similar or identical to the factors reported in [5]. From thereon, factors from [5] were used when applicable. The result was a set of stable and common categories that constitute candidate maintainability aspects. Each aspect was examined for coherence and strength across the cases based on Eisenhardt's recommendations for analyzing within-group similarities coupled with intergroup differences [39]. Some candidate aspects were not replicated across cases, and are therefore not included in the final results.

To analyze the impact of the identified factors, a cross-case matrix was used to compare the factors across cases based on the previous maintainability evaluation and the perception of the maintainers. Finally, each factor was analyzed individually, alongside the statements from the developers that were grouped together, in order to determine the degree to which this factor can be reflected based on the definitions of the twenty-two code smells described by Beck & Fowler [4] and the design principles described by Martin [40].

[9]http://www.transana.org

Table III
CROSS-CASE MATRIX OF MAINTAINABILITY FACTORS AND SYSTEM EVALUATIONS

Factor	defined in [5]	#statements (#dev)	A	B	C	D
Appropriate technical platform	Yes	19 (6)	2/2 Neg^0	3/3 Neg^2	2/2 Neg^2	1/2 Neg, 1/2 Pos^1
Coherent naming	Yes	3 (2)	1/1 Neg^0	Nm	1/1 Neg^0	Nm
Design suited to the problem domain	Yes	6 (3)	Nm	3/3 Neg^1	Nm	Nm
Encapsulation	Yes	3 (3)	2/2 Pos^0	Nm	1/1 Pos^0	Nm
Inheritance	Yes	1 (1)	Nm	1/1 Neg^2	Nm	Nm
(Proprietary) Libraries	Yes	4 (2)	Nm	2/2 Neg^2	Nm	Nm
Simplicity	Yes	21 (6)	3/3 Pos^1	3/3 Neg^2	3/3 Pos^1	1/1 Neg^1
Three-layer architecture	Yes	6 (4)	1/1 Neg^1	Nm	2/2 Neg^2	2/3Pos, 1/3 Neg^2
Use of components	Yes	4 (3)	Nm	1/1 Neg^1	2/2 Neg^2	1/1 Pos^2
Design consistency	No	27 (6)	3/3 Neg	2/3 Pos,1/2 Neg	3/3 Neg	3/3 Pos
Duplicated code	No	2 (2)	2/2 Neg	Nm	Nm	Nm
Initial defects	No	5 (3)	Nm	1/1 Pos	2/2 Neg	2/2 Pos
Logic Spread	No	3 (2)	1/1 Neg	2/2 Neg	Nm	Nm

IV. RESULTS

Comparing expert assessment and developer impression: From the cross-case synthesis and axial coding, thirteen maintainability factors emerged. Nine of these confirm earlier findings by Anda [5], and four new factors were identified. Table III shows the factors that emerged, the number of statements or quotes associated to each factor, and the number of developers who made statements on these factors (this last value enclosed in a parenthesis). Note that the factors "Comments" and "Standard naming conventions" from Anda are not included in our list because only one developer mentioned the first and none mentioned anything related to the second factor.

The last four columns of Table III show the developers' perception of each factor for every system. The coding is as follows: "M/N Neg" means that N developers talked about that factor in the system and M of them had a negative impression; "M/N Pos" is similar for a positive impression. Moreover, since the perceptions are based on what was mentioned in the interviews, some aspects are not covered for every system, in which case they are marked with "Nm" (not-mentioned). As an example, consider the factor Three-layer architecture: three developers mentioned this for system D, two of them had a positive impression, and one was negative; no developers mentioned this topic for system B (hence "Nm"). Finally, for the factors defined in the work by Anda, the superscript values indicate the degree of matching between the expert evaluation and the developers' impression where 0=no match, 1=medium match, and 2=full match.

If we observe the number of references to maintainability factors, we can distinguish Technical platform, Simplicity and Design consistency as the factors most mentioned by all six developers. The factors with most matches are on the negative impressions on system B, in particular to Technical platform, Design suited to the problem domain, and Simplicity. Systems A and C display a high degree of agreement on positive impression over the Simplicity of the systems. System A displayed the highest rate of disagreements between expert judgment and the maintainers, and System D displayed the highest degree of agreement, both parties considering this system as fairly good.

Relating maintainability factors to code smells: Next, for each maintainability factor identified in our analysis, we discuss how it was perceived to affect maintenance, and analyze which code smells relate to it. For factors that cannot be related to code smells, we discuss which alternative methods can be used to evaluate them.

Appropriate technical platform – This factor manifested in several forms across the projects. In System B, it appeared in the form of a complex, proprietary Java persistence framework, a particular type of authentication based on Apache Tomcat Realm, and a memory caching mechanism (which became obsolete with the new Simula CMS). Developers claimed that they spent many hours trying to understand each of these mechanisms. One of the experts in [5] stated: *"Many problems with systems maintenance are related to undocumented, implicit requirements that surface when a system is moved to a different environment"*. Here, this was evidenced via two widely used, restrictive interfaces. Both were made under the assumption that identifiers for objects would always be Integers. However, in the new environment, String type object identifiers were needed. Interface replacement was not possible since the implementation was based on primitive types instead of of domain entities.

For systems A and C, this factor manifested in the lack of appropriate persistence mechanisms, resulting in very complex SQL queries embedded in the Java code. Developers saw the integration of Simula's CMS and the SQL queries as one of the biggest challenges. Another example in System C was the log mechanism, which did not stream out the standard error messages generated by Tomcat. As a result, developers were forced to introduce try and catch statements in many segments of the Java and JSP code. The variation in these cases shows that it is difficult for code smells to reflect such situations, instead requiring expert evaluations.

Coherent naming – This factor reflects to how well the *code vocabulary* represents the domain, and how it facilitates

Table IV
MAINTAINABILITY FACTORS AND THEIR RELATION TO CURRENT DEFINITIONS OF CODE SMELLS

Factor	Covered by code smell	Code smells associated	Autom. smell detection	Alternative evaluation
Appropriate technical platform	no	NA	no	Expert judgment
Coherent naming	no	NA	no	Semantic analysis, Manual inspection
Design suited to problem domain	partially	Speculative Generality	no	Expert judgment
Encapsulation	partially	Data Clump	partially	Manual inspection
Inheritance	partially	Refused Bequest, Simulation of multiple inheritance	partially	Manual inspection
(Proprietary) Libraries	partially	Wide Subsystem Interface	partially	Expert judgment, Dependency Analysis
Simplicity	partially	God Class, God Method, Lazy Class, Message Chains, Long Parameter List	yes	Expert judgment
Three-layer architecture	no	NA	no	Expert judgment
Use of components	partially	God Class, Misplaced Class	yes	Semantic analysis, Manual inspection
Design consistency	partially	Alternative Classes with Different Interfaces, ISP Violation, Divergent Change, Temporary Field	partially	Semantic analysis, Manual inspection
Duplicated code	yes	Duplicated code, Switch statements	yes	Manual inspection
Initial defects	no	NA	no	Acceptance tests, Regression testing
Logic Spread	partially	Feature Envy, Shotgun Surgery, ISP Violation	yes	Manual inspection, Dependency analysis

the mapping between domain and requirements and code. Examples include System A where a developer did not understand why a class was called "StudySortBean" when its responsibility was to associate a Study to an Employee. In System C, similar situations occurred at the method level (*"One of the most problematic factors was strangely named methods"*). The meaningfulness of a code entity's name cannot be evaluated by code smells, and may require manual inspection and/or semantic analysis.

Design suited to the problem domain – Relates to selecting a design that is adequate for the context of the system. Experts stated: *"...the complexity of the system must justify the chosen solution, and the maintenance staff must be competent to implement a solution in accordance with the design principles"*. System B shows a counter-example, where a complex proprietary persistence library was used instead of a generic one, better suited to small/medium sized information systems. This relates to the Speculative Generality smell. However this factor (and smell) is very difficult to evaluate via automated code analysis and requires expert judgment.

Encapsulation – Experts in Anda's study concluded that small container classes were used to deliver more than one object as output from methods. They indicated that this introduces dependencies that can lead maintenance problems. In general encapsulation aims to hide the internals of a class to prevent unauthorized access. Developers perceived that Systems A and C had acceptable encapsulation, which resulted in a localized ripple-effect. Code smells such as Data Clumps are indictors of inadequate encapsulation, and they can be complemented with manual inspection.

Inheritance – In System B, multiple interfaces were used to simulate multiple inheritance. However, this practice led to such complex (and dynamic) dependencies between classes that it prompted one of the developers to remove code that he erroneously considered "dead code". After finding out that it wasn't, considerable effort was needed to roll back this change. Currently, there are no code smell definitions related to "Simulation of multiple inheritance" but we propose it as a new smell, considering the serious consequences this factor could entail. Given the small/medium size of the other systems, inheritance was not used extensively, and this characteristic did not manifested in the interviews. In addition to "Simulation of multiple inheritance", Refused Bequest ("Subclasses don't want or need everything they inherit") can also be useful to evaluate this factor.

(Proprietary) Libraries – As mentioned before, System B contained a complex proprietary library that transforms logical statements to queries for accessing the database. References to this persistence logic were scattered over the system, forcing the developers to inspect a considerable amount of files in order to understand and use the library, especially for Task 3. The experts in Anda's work indicate that *"the use of libraries may imply a greater amount of code, which in itself is less maintainable. The use of proprietary libraries may imply lower maintainability, because new developers will need to familiarize themselves with them."* This was exactly the case in System B. Although such libraries can be analyzed with static analysis, their maintainability implications depend on how they were used previously and the proportion of the system that relies on them. A design principle violation that relates to this factor is the Wide Subsystem Interface: "A Subsystem Interface consists of classes that are accessible from outside the package they belong to. The flaw refers to the situation where this interface

is very wide, which causes a very tight coupling between the package and the rest of the system" [40]. Evaluation approaches are dependency analysis and expert judgment.

Simplicity – Simplicity was considered a very important factor by both experts and maintainers, and clearly distinguishes our four systems. Systems A and C were perceived as simple and fast to get started with, in contrast to System B, which was perceived as extremely complex, in particular due to the number of code elements and the interconnections between them. The large number of classes required time to understand the code at higher level, and to find relevant information. Experts and developers agreed that there was a large proportion of "empty classes" in System B, making it difficult to identify the pertinent ones. A very descriptive remark by one developer was *"I spent more time for understanding than for coding."* Another dimension of this factor is the size of the classes and methods. For all four systems, developers complained about at least two classes "hoarding" the functionality of the system. These were extremely large in comparison to the other classes and displayed high degrees of afferent and efferent coupling dispersion.[10] Referring to this factor, a developer stated: *"Size of methods and classes is important, because I need to remember after reading a method what was it about!"*

There are a number of code smells that relate to this factor: God Class and God Method, and Long Parameter List can identify cases as the one previously described. Lazy Class can find the "empty" redirection classes mentioned by the experts. Traditional metrics such as LOC, NOC can be useful to assess this factor as well. Finally, Message Chains can indicate complex, long, data/control flows that typically result from such redirections.

Three-layer architecture – This factor manifested in System C, which had excessive business logic embedded in JSP files. This forced the developers to work with the logic in the JSP files, performing modifications in a "manual" way, as they were deprived from much of the functionality in Eclipse that was only available for Java files. In System D, developers were slightly taken aback due this system having an additional layer inside the business logic layer (enabling the web presentation layer to be replaced by a standalone library in the future), although this was not considered very problematic. Code smells do not reflect this level of abstraction, thus alternative approaches are needed to evaluating this factor. Several approaches have been proposed, most of them relying to a certain degree on human input [41].

Use of components – The organization of classes should be according to functionality, or according to the layer of the code on which they operate. This factor played a role in System C which lacked a clear distinction between business and data layers. The "hoarders" mentioned earlier are orthogonal manifestations of this factor, since classes begin to grow because they cover more functionality than they should. Although semantic aspects of this factor (such as the nature of functionality consistently allocated to corresponding classes) should be evaluated separately, the quantitative perspectives (there should not be classes that do too little or too much) on this factor can be evaluated by code smells such as God Class. In addition, Misplaced Class could be used to identify outliers that are in the wrong layer or package. Given that this factor requires considerable semantic knowledge, it cannot be addressed solely by code smells.

Design consistency – This factor was the one mentioned most by developers during the interviews and was not covered by experts. It refers to the impossibility for developers to oversee the behavior of the system, or the consequences of their changes, because they were constantly facing contradictory or inconsistent evidence. The inconsistencies in System C manifested both at the variable level (*"I had troubles with bugs in DB class, from mistakes in using different variables"*) and at the design level (*"Design in C was not consistent, similar functionality was implemented in different ways"*).

Confusing elements also occurred in system A, where the data/functionality allocation was not semantically coherent, nor consistent (*"Data access objects were not only data access objects, they were doing a lot of other things"*). This resulted in false assumptions about the system's behavior, and developers would get confused when they found evidence that contradicted earlier assumptions. Alternatively, when the false assumptions were not confuted, they led to the introduction of faults (*"The biggest challenge was to make sure changes wouldn't break the system; because things were not always what they seemed to be"*).

A contrasting example was system D, which was perceived as very consistent by all three developers who worked with it. Quotes from the developers illustrate this:

- *"It was about applying the same pattern of changes for similar classes"*
- *"There were no surprises, if I change the class X, I could follow the same strategy to change class Y"*
- *"If something breaks, in system D was easier to trace the fault"*

Developers working with system D indicated that having a consistent schema for variables, functionality and classes was a clear advantage for code understanding, information searching, impact analysis and debugging.

Most of the observed cases were of semantic nature, which were combined with structural-related factors, thus evaluating this factor constitutes a rather challenging task. We suggest a couple of code smells that can potentially help to identify design inconsistencies, as described in Table V.

In addition, ISP Violation can be an indirect indicator of inconsistency. Martin [40] states that: "Many client specific interfaces are better than one general purpose interface".

[10] Afferent coupling spread or dispersion denoted many elements having dependencies on one element, which is typical of widely used interfaces. Efferent coupling spread or dispersion denotes one element depending on many interfaces.

Table V
CODE SMELLS RELATED TO THE IDENTIFICATION OF DESIGN INCONSISTENCIES

Code Smell	Description
Alternative Classes with Different Interfaces	Classes that mostly do the same things, but have methods with different signatures
Divergent Change	One class is commonly changed in different ways for different reasons
Temporary Field used for several purposes	Sometimes temporary variables are used across different contexts of usage within a method or a class.

When "fat interfaces" start acquiring more and more responsibilities and start getting a wider spectrum of dissimilar clients, this could affect analyzability and changeability. The presence of ISP Violation by itself does not imply the presence of design inconsistencies, but one can reasonably assume that chances of finding inconsistencies in an wide interface is higher than interfaces that are used only by a small set of clients. Alternative options to evaluate this factor can be semantic analysis, and manual inspection.

Duplicated code – The maintainability of a system can be negatively affected by including blocks of code that are very similar to each other (code clones). In Anda's work, the experts include this as an aspect of Simplicity [5]. In our study, we identified cases where Simplicity and Duplicated code manifested as separate factors, which is why we add this factor as an independent one. The developers stated that System A contained many copy-paste related ripple-effects and they considered duplication as one of the biggest difficulties. Duplicated code has attracted considerable attention from the research community and various approaches exist for detection, removal and evolution in the presence of clones. An additional related smell is Switch Statements where conditionals depending of type lead to duplication.

Initial defects – A factor affecting maintainability that was not mentioned by the experts in Anda's work is the amount of defects in the system. In the case of System C, developers unanimously complained about how many defects the system contained *at the beginning of the maintenance phase*. They claimed that they spent a lot of time correcting defects before they could actually complete the tasks. Systems D and B were perceived as not having many initial defects. Although some work has been done for predicting the defect density at class level using code smells [28], in general this factor needs to be evaluated by other means, such as a set of regression tests, acceptance tests, etc.

Logic Spread – Maintainers of system B mentioned that it contained a set of classes whose methods made many calls to methods or variables in other classes, and that they were forced to examine all the files called by these methods, resulting in considerable delays. A developer mentioned: *"It was difficult in B to find the places to perform the changes because of the logic spread"*. Although the factor Simplicity is related to this factor, Simplicity only covers the amount of elements in a system, and not the number of interactions between them. Logic spread has been addressed early by metrics like afferent and efferent coupling. Smells such as Feature Envy indicate efferent coupling dispersion. Other smells such as Shotgun Surgery and Interface Segregation Principle Violation focus on afferent coupling dispersion, consequently they can be useful to evaluate this factor as well. Moreover, it is related to the scattering and tangling that is typically associated with implementing *crosscutting concerns* in non-aspect-oriented languages [42].

V. DISCUSSION

Comparison of factors across sources: Thirteen factors were identified during the maintenance project, nine of them coinciding between experts and maintainers. This supports the findings in [5]. Yet, some factors were mentioned more often than others by the developers. For instance, factors such as Standard naming conventions and Comments did not play such an important role for developers. Discrepancies can be due to the fact that experts may lack enough contextual information at the time of the evaluation to weigh the impact or importance of a given factor accordingly. Developers in the other hand are conditioned by their maintenance experience, so factors such as the nature of the task may play an important role over what is perceived as the most relevant of the software factors.

Conversely, all developers who participated in the project unanimously mentioned factors such as Appropriate technical platform and Simplicity. This may indicate that such factors should be given more attention, since they may play an important role regardless of the maintenance context. This result also suggests that to achieve more accurate maintainability evaluations based on expert judgment, the usage of enough contextual detail may be required to enable experts the priorization of certain factors over others. One example of such approach is the usage of "maintenance scenarios" proposed by Bengtsson and Bosch [43].

Some factors identified during the project were not reported previously, and they can complement the findings from [5]. Factors such as Design consistency and Logic Spread were perceived as very influential by the maintainers, but were not mentioned by the experts. This can be due to the fact that system maintainability evaluations based on expert-judgment focus on factors at higher abstraction levels, as opposed more fine-grained factors observed by developers. Developers will necessarily capture different factors than experts do, because they are the ones having to dive into the code and suffer the consequences of different design flaws. In addition, most expert evaluations are time-constrained, which would not allow the experts to go into many details. The multiplicity of perspectives observed in this study reflects

on previous ideas about the need of diversity in evaluation approaches in order to attain more complete pictures of maintainability.

Scope/capability of code smells: Situations described by the developers during the interview provided a clear-cut outlook on the difficulty of using code smell definitions for analyzing certain maintainability factors. Such was the case for factors as Appropriate technical platform, Coherent naming, Design suited to the problem domain, Initial defects and Architecture. These factors would need additional techniques to be assessed. Yet, it is worth noting that factors related to code smells include both factors addressed by traditional code metrics as well as factors mainly addressed by expert judgment. For instance, some code smells can support the analysis of factors such as Encapsulation and Use of components (which according to [5] are not captured by software metrics). Factors as Simplicity, which are traditionally addressed by static analysis means, are closely related to the code smells suggested in Table IV. Moreover, several detection techniques for these smells are based on size-related software metrics. Logic spread factor also is largely related to the notion of coupling and cohesion initially described in [44]. These results hint at the potential of code smells to cover a more heterogeneous spectrum of factors than software metrics and expert judgment individually.

An interesting factor identified throughout the study was Design consistency, which according to developers played a major role during different maintenance activities (e.g., understanding code, identifying the areas to modify, coding, debugging). This factor was interpreted in a broad sense, crosscutting abstraction levels (e.g., consistency at variable, method, class level) and was not limited to the "naming" of elements. Despite this rather broad and inclusive definition, we see great potential for a number of code smells that each can help to identify a certain subset of inconsistencies.

For the identified maintainability factors, we find that eight of them are addressable by current code smell definitions. However, in most cases these code smells would need to be complemented with alternative approaches such as semantic analysis and manual inspection. The suggestions on code smells presented in this work were derived theoretically, based on definitions of code smells available in the literature, and as such, they should be treated as suggestions or starting points for further empirical studies to validate their usefulness. We make no claims concerning the degree or descriptive richness of a smell in relation to a maintainability factor, since that would fall out of the scope of this work. Moreover, some code smells are not detectable via automated means, so even though they reflect certain maintenance factors, other means are needed to assess these factors.

Finally, these results are contingent on the nature of the tasks and characteristics of the maintenance project, for which are suggested as a preliminary set of factors. Replications of this study in different industrial contexts are needed in order to extend and support our findings.

Threats to validity: Following a grounded theory approach, the most important quality aspects of our findings are their *fit* and *relevance* [45]. For the fit perspective we argue that the different software maintainability factors were consistently assigned to different categories, most of them common knowledge in the software engineering research community. With respect to relevance, we argue that most factors were relevant to both the experts who evaluated the systems, and the developers who maintained it. We addressed the construct validity threat with data-triangulation and investigator triangulation. With respect to internal validity, we argue that most of our results are descriptive, although it relies on the interpretation of the researcher who carried out the interviews. Transcripts from the daily interviews compensate for any potentially concealed issues that developers did not want to discuss during the open-ended interviews. With respect to external validity, the results are contingent to several contextual factors, such as the systems (i.e., medium-size Java web information systems), the nature and size of the tasks, and the project modality (i.e., solo-projects). Finally, one can argue on the representativeness of the maintenance tasks carried out, although all of them were based on real needs and their duration and complexity are representative of real-life projects.

VI. Conclusion and Future Work

By understanding the capability and limitations of different evaluation approaches to address different factors influencing software maintainability, we can achieve better overall evaluations of maintainability. Code smells can provide insight on different maintainability factors which can be improved via refactoring. However, some factors are not reflected by code smells and require alternative approaches to evaluate them.

This paper describes a set of factors that were identified as important for maintainability by experts who evaluated the four Java systems in our study [5], and by six developers who maintained those systems for 14 days while implementing a number of change requests. Through our analysis, we identified some new factors not reported in previous work [5], which were perceived as important by the developers.

Based on the explanations from the developers, we found that some of the factors can potentially be evaluated (at least partially) by using some of the current code smell definitions. The contributions of this paper are three-fold: (1) We confirm and complements the findings by Anda [5], by extracting maintainability factors that are important from the software maintainer's perspective, and (2) Based on the manifestations of these factors during an industrial maintenance project, we identify which code smells (or alternative analysis methods) can evaluate them, and (3) We provide an overview of the capability of code smell definitions to evaluate the overall maintainability of a system.

Given the fact that there are many important factors not addressable by static analysis, and that not all code smells are actually automatically detectable, we agree with the statements from Anda [5] and Pizka & Deissenboeck [19] that there is a need for combining different approaches in order to achieve more complete, and accurate evaluations of overall maintainability of a system.

Future work includes detailed analysis of problems reported by developers during the maintenance project and associate them with code smells detected automatically within the code. This can to provide a quantitative perspective in relation to the coverage-level or capability of code smells to uncover problematic code.

REFERENCES

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

[2] T. C. Jones, *Estimating software costs*. McGraw-Hill, 1998.

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2005.

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

[5] B. Anda, "Assessing Software System Maintainability using Structural Measures and Expert Assessments," in *IEEE Int'l Conf. Softw. Maintenance*, 2007, pp. 204–213.

[6] M. Alshayeb and L. Wei, "An empirical validation of object-oriented metrics in two different iterative software processes," *IEEE Trans. Softw. Eng.*, vol. 29, no. 11, pp. 1043–1049, 2003.

[7] H. Benestad, B. Anda, and E. Arisholm, "Assessing Software Product Maintainability Based on Class-Level Structural Measures," in *Product-Focused Softw. Process Improvement*, 2006.

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

[9] J. A. McCall, P. G. Richards, and G. F. Walters, *Factors in Software Quality*. NTIS, 1977, vol. I.

[10] J. C. Granja-Alvarez and M. J. Barranco-García, "A Method for Estimating Maintenance Cost in a Software Project," *J. Softw. Maint.*, vol. 9, no. 3, pp. 161–175, 1997.

[11] H. Leung, "Estimating maintenance effort by analogy," *Empirical Software Engineering*, vol. 7, no. 2, pp. 157–175, 2002.

[12] J. Mayrand and F. Coallier, "System acquisition based on software product assessment," in *Intl Conf. Softw. Eng.*, 1996.

[13] T. Rosqvist, M. Koskela, and H. Harju, "Software Quality Evaluation Based on Expert Judgement," *Software Quality Control*, vol. 11, no. 1, pp. 39–55, 2003.

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

[15] T. McCabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

[16] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier, 1977.

[17] K. D. Welker, "Software Maintainability Index Revisited," *CrossTalk - J. of Defense Software Engineering*, 2001.

[18] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, 1994.

[19] M. Pizka and F. Deissenboeck, "How to effectively define and measure maintainability," in *Softw. Measurement European Forum*, T. Dekkers, Ed., 2007.

[20] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Working Conf. Reverse Eng.*, 2001.

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

[22] W. C. Wake, *Refactoring Workbook*. Addison-Wesley, 2003.

[23] M. Mäntylä, "Software Evolvability - Empirically Discovered Evolvability Issues and Human Evaluations," PhD Thesis, Helsinki University of Technology, 2009.

[24] R. Marinescu, "Measurement and quality in object-oriented design," in *Int'l Conf. Softw. Maint.*, 2005, pp. 701–704.

[25] E. H. Alikacem and H. A. Sahraoui, "A Metric Extraction Framework Based on a High-Level Description Language," in *Working Conf. Source Code Analysis and Manipulation*, 2009.

[26] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *Working Conf. Reverse Eng.*, 2009.

[27] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[28] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *J. Syst. Softw.*, vol. 80, no. 7, 2007.

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

[30] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *J. Syst. Softw.*, vol. 65, no. 2, pp. 127–139, 2003.

[31] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Int'l Conf. Softw. Maint.*, 2008.

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

[33] S. M. Olbrich, D. S. Cruzes, and D. I. Sjoberg, "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.*, 2010, pp. 1–10.

[34] B. C. D. Anda, D. I. K. Sjøberg, and A. Mockus, "Variability and Reproducibility in Software Engineering : A Study of Four Companies that Developed the Same System," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 407–429, 2009.

[35] G. G. R. Bergersen and J.-E. Gustafsson, "Programming Skill, Knowledge, and Working Memory Among Professional Software Developers from an Investment Theory Perspective," *J. of Individual Differences*, vol. 32, no. 4, pp. 201–209, 2011.

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

[37] A. Strauss and J. Corbin, *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[38] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" Simula Research Laboratory, Technical Report (2012-10), 2012.

[39] K. M. Eisenhardt, "Building Theories from Case Study Research," *Academy of Management Review*, vol. 14, no. 4, 1989.

[40] R. C. Martin, *Agile Software Development, Principles, Patterns and Practice*. Prentice Hall, 2002.

[41] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures*. Addison-Wesley, 2001.

[42] M. Marin, A. V. Deursen, and L. Moonen, "Identifying Crosscutting Concerns Using Fan-In Analysis," *ACM Trans. Softw. Eng. Meth.*, vol. 17, no. 1, 2007.

[43] P. Bengtsson and J. Bosch, "Architecture level prediction of software maintenance," *European Conf. Softw. Maint. and Reeng.*, pp. 139–147, 1999.

[44] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Softw. Eng.*, vol. 20, no. 3, 1994.

[45] J. Shanteau, "Competence in experts: The role of task characteristics," *Organizational Behavior and Human Decision Processes*, vol. 53, no. 2, pp. 252–266, 1992.

APPENDIX

Table VI
EXCERPT OF STATEMENTS FROM DEVELOPERS, ASSOCIATED TO DIFFERENT MAINTAINABILITY FACTORS

System	Statement	Factor
A	"A is a simple system and it was fast to start working with"	Simplicity
	"Big challenge to make sure changes wouldnt break the system; things were not always as they seemed to be"	Design consistency
	"It has too much freedom or is too arbitrary with respect to the design"	Design consistency
	"It was quite fast to learn and understand because it was not too complex"	Simplicity
	"Data access objects were not only data objects, they were doing a lot of other things"	Design consistency
	"Bugs were proportional to the changes in a class, but they will stay within the class"	Encapsulation
B	"Spent long time extending it because there were so many manual changes"	Technical platform
	"Spent long time learning how the system worked"	Simplicity
	"There was a serious lack of flexibility with the framework for generating the reports"	Technical platform
	"I was constantly changing the persistence layer, requiring a lot of coding"	Technical platform
	"The interface was too limited"	Technical platform
	"Spent quite a lot of time in understanding a library for generating SQL queries"	Libraries
	"It was a lot of rework, had to go back and change things because of the framework"	Technical platform
	"The main problem in B were the identifiers for people and publication"	Technical platform
	"Task 1 was easier in A than in B because of the construction of queries in B"	Technical platform
C	"The system is maintainable because its small"	Simplicity
	"Had to rework the queries in the main class several times"	Technical platform
	"System was full of bugs"	Initial defects
	"The bugs will only break one screen"	Encapsulation
	"C is straightforward, easy to start working with"	Simplicity
	"I had troubles with bugs in DB class, and mistakes in using different variables"	Design consistency
	"In C one big class sometimes made it hard to find the right method"	Use of components
	"A messy system but not that complicated, so easy to learn"	Simplicity
	"Everything had to be done from scratch"	Technical platform
	"Most problematic java element: lack of comments and strangely named methods"	Coherent naming
D	"Some extra work for the additional layer with Handlers, but in general it looked good"	Architecture
	"System is balanced, no classes or methods that do too much"	Use of Components
	If I change the class X, I could follow the same strategy to change class Y	Design consistency
	"Spent 10 times more learning this system than the previous, small system"	Simplicity
	"There were not many bugs"	Initial defects