Assessing Software System Maintainability using Structural Measures and Expert Assessments

Bente Anda
Simula Research Laboratory and University of Oslo
bentea@simula.no

Abstract

Software maintenance is often expensive; hence, strategies for assessing the maintainability of complete software systems are important. Nevertheless, a software client usually has few means of assessing the maintainability of a software system as part of the acquisition process. Assessing the maintainability of complete systems is difficult due to the influence of many factors, such as the people, tasks and tools, in addition to the code. Furthermore, most research on maintainability focuses on individual classes of individual systems.

This paper describes an empirical study in which the maintainability of four functionally equivalent systems developed in Java was assessed using both structural measures and expert assessments. The results suggest that such a combination may be useful. Although the assessment based on structural measures mostly corresponded with the expert assessments, there were several examples of potential problems regarding maintainability that were not captured by the structural measures.

1. Introduction

Due to the high costs of software maintenance, a software client wants systems that will be easy to maintain. However, software clients typically have few means of assessing the maintainability of the software system that they are in the process of acquiring. It is also far from common for software development organizations to have control over the maintainability of their developed software. Furthermore, even if the software developers manage to assess the maintainability of their software, they may not wish to share this information with their clients. From the point of view of a software development organization, it may be most economically beneficial to spend few resources on developing maintainable and extendable software initially, and then achieve higher earnings from providing costly extensions in the maintenance phase.

It is therefore of great economic importance for software clients to be able to assess the maintainability of software systems or products. Nevertheless, what constitutes a maintainable software system is not well defined, and we do not know the conditions under which a system is maintainable. To the author's knowledge, few studies have compared the maintainability of different systems.

Maintainability may be affected by a large number of factors in addition to properties of the code, such as the qualifications of the people performing the maintenance, the maintenance tasks, and the tools used. Furthermore, the fact that technology changes frequently means that new standards for design and architecture of software systems are set frequently. It may therefore not be feasible to establish once and for all the factors that affect the maintainability of software systems.

Assessing the maintainability of a software system is equivalent to making a prediction, on the basis of information about the existing system, about the effort that will be expended on maintaining it. The strengths and weaknesses of experts, formal methods, and combinations of these have been studied in the field of software estimation. The findings are that a combination of expert assessments and formal methods usually provides the best results [12]. Although there are obvious differences between estimating software development effort and assessing maintainability (maintenance effort), there are also similarities, which indicates that further studies on the role of expert assessments may be beneficial also in the field of software maintainability.

This paper presents a qualitative comparison of assessments of maintainability based on structural measures and expert assessments. Four functionally equivalent systems developed using Java, with Simula Research Laboratory as client, provided us with a unique opportunity to compare the effects of different design choices on maintainability. The goals of the study were twofold: 1) to identify properties of Java software systems that are considered important for maintainability by experts who have a great deal of experience of Java development, and 2) to identify strengths and weaknesses of assessing maintainability using structural measures and using expert assessments.

The results showed that there was mostly correspondence between the overall assessments of maintainability based on structural measures and those based on expert assessments, but also that many potential maintainability problems are difficult to detect using only structural measures. Examples of such maintainability problems are: choice of concepts to be implemented as classes that do not support understanding of the code and/or do not support traceability from the requirements; trivial components and unnecessary classes; a design that is not appropriate for the size and complexity of the system; and the inappropriate use of names. These problems are difficult to detect using structural measures, therefore the results support the claim that the maintainability of software systems can best be assessed by using a combination of expert assessments and structural measures. The results of the study reported herein could constitute one step towards formulating a combined strategy for assessing the maintainability of complete software systems.

The remainder of this paper is organized as follows. Section 2 describes the concept of maintainability and discusses factors that affect maintainability. Section 3 presents approaches to assessing maintainability. Section 4 describes the case study. Section 5 describes the use of structural measurements when assessing the maintainability of the case systems. Section 6 describes the expert assessments. Section 7 describes the scope of the results and presents threats to validity. Section 8 concludes and presents plans for future work.

2. Maintainability

The IEEE standard for software engineering terminology defines software maintainability as the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment [11].

The maintainability of a software system is affected by design and architectural principles. However, such principles change over time as technology changes; hence, the establishment of definitive criteria for assessing maintainability is probably not feasible. Nevertheless, many criteria are likely to be stable for quite some time; hence, the fast pace in the software industry should not be used as a reason to abandon the attempt to improve strategies for assessing the maintainability of complete software systems.

Two project factors affecting maintainability are: the maintenance tasks to be performed on the system and the people who will perform them. With respect to the former, small tasks restricted to a limited part of the system may be easier to perform on software with fewer and larger classes because it is then easy to identify the class or cluster of classes where changes must be made. By contrast, larger change tasks that require changes to larger parts of the software depend more on the organization of the code and the programming principles that have been applied.

With respect to the latter, the IT industry is characterized by people who frequently change their job, or at least change projects, within an organization. Hence, new developers are often assigned to an existing project and must relate to the old code in order to develop the system further. A maintainable system should therefore be easy to understand for new developers. Furthermore, studies have shown that the experience and education of the developers play a significant role in understanding and applying the principles behind different designs [3,18].

3. Assessing Maintainability

Maintainability can only be measured indirectly. When assessing maintainability we typically must choose between using (i) well-defined measures that may not correspond to our intuitive understanding of maintainability and that may use, for example, structural code measures, and (ii) a vague definition of maintainability and use experts.

3.1 Structural Measures

A large amount of empirical research has been done on measuring the structural properties of software. Briand and Wuest provide an overview of empirical work on structural measures and conclude that measures of size, coupling and cohesion of classes are generally correlated with their maintainability [6].

Chidamber & Kemerer's set of structural measures, denoted CK-metrics, is probably the most used. Its relation to maintainability has been confirmed empirically [7,8]. The CK-metrics include the following class-level measures: WMC (Number of methods in class), CBO (Coupling between objects), NOC (Number of children), DIT (Depth of inheritance tree), LCOM (Cohesion of methods) and RFC (Response set for a class).

Research on maintainability has focused mostly on the maintainability of classes or clusters of classes of individual systems, while the maintainability of complete software systems has received relatively little attention. Furthermore, models for assessing maintainability that are based on structural measures have rarely been used on unknown systems, that is, on systems to which they have not been calibrated [4]. However, Bakota et al. investigated how the different CK-metrics behaved over a set of unrelated systems. They found that the measures WMC and RFC behaved very differently on different systems.

Another set of metrics, MOOD, is intended to provide an overall assessment of a software system [10]. The set measures method hiding, attribute hiding, method inheritance and attribute inheritance, coupling, and polymorphism. Hence, it is focused more on methods than on classes, as is the case for the CK-metrics. However, the relationship of the MOOD set of metrics to maintainability of complete software systems has not been confirmed empirically.

3.2 Expert Assessments

The most commonly used strategies in practice for assessing maintainability are guided and unguided expert assessments [15]. One example of a guided strategy is The Air Force operation Test and Evaluation Center (AFOTEC) pamphlet, which provides a rich set of instructions for evaluating software maintainability [1]. Another guided strategy is to search the code for so-called code smells. According to Fowler and Beck, a defined set of code smells can indicate poor maintainability and a need for refactoring [9]. However, this has not been confirmed empirically and in an experiment experts judged the code differently with respect to the presence of code smells [15].

The reliability of expert assessments is open to question. Different studies have obtained different results regarding the reliability of expert assessments. Schneiderman found little agreement in the expert evaluation of code quality where experts had not developed the code [16], while Shepperd found high reliability within development teams [17].

Another difficulty with expert assessments is that they depend on having people who are both qualified to make them and are representative of, or understand the qualifications of, those who will perform maintenance on the systems. Finally, experts may also be biased in their opinions, for example by considering designs that they are not familiar with as problematic.

3.3. Combined Assessments

There are few studies on the correspondence between measurement-based assessments and expert assessments or on how to combine measurement strategies. Mayrand and Coallier describe an approach for software system assessment used as part of an acquisition [14]. This approach combines structural measures with expert assessments, but it is described in little detail. Measurements were found useful to guide design evaluations in [13].

Assessing the maintainability of a software system is equivalent to predicting future costs related to maintaining the system. The strengths and weaknesses of using expert judgment vs. formal methods in making predictions in software engineering have been studied in the context of software estimation [12]. Results from that field show that expert assessments are more common in practice than formal methods and also that expert assessments outperform the formal methods. The reasons for this are suggested to be that the important variables influencing development effort are not well established, the relationship between characteristics of a software project, and the formal methods usually only have small sets of previous projects on which they can be calibrated. However, expert assessments may be biased due, for example, to the expectations of project members or simply the current mood of the estimator(s). Hence, combining expert estimates and formal estimation models appears to yield the best results in software estimation [12].

There are both similarities and differences between software estimations and assessments of maintainability. Both are about assessing future costs, but the main difference is that assessing maintainability is about assessing software that already exists. However, the results from software effort estimation suggest that it is worthwhile to investigate how to combine expert assessments and more formal assessments based on, for example, structural measures, in the field of software maintenance as well.

A problem with expert assessments of large software systems is that it will often be impossible to assess all the code. A strategy for selecting which parts of the code to assess is therefore required. AFOTEC, relies on random samples of the code [1], while the approach described by Mayrand and Coallier relies on samples chosen by the developers [14].

4. The Case Study

The Software Engineering (SE) Department at Simula Research Laboratory sent out a tender for a software system to 85 Norwegian software consultancy firms. Of the 35 companies that responded, four companies were selected to develop individual software systems based on the same requirements specifications. These software systems, hereafter called A, B, C and D, presented us with the challenge of assessing which was the most maintainable, and also provided us with a unique opportunity to investigate and compare the maintainability of four functionally equivalent systems. The development projects and the resulting systems are described more in detail in [2,5]. The four companies and the people involved in the development knew that they were participating in a research project and agreed to it.

The system to be developed was a web-based system for handling the studies conducted by the SE department at Simula Research Laboratory. The functional requirements were described in detail to ensure functionally equivalent systems (this was also ensured through detailed acceptance tests conducted by the client, us). The business logic was simple and the requirements specification did not include specific requirements on the quality of the code in terms of, for example, maintainability or reusability. Furthermore, the system to be developed had no characteristics that would make a specific design strategy right or wrong. Each company chose their own development process and what emphasis they would place on code quality and maintainability.

After finishing the development, the teams provided their own opinions on the quality of the code in interviews (see Table 1).

Table 1. Developers' assessment of maintainability

Company	Opinion
A	Maintainability is acceptable. The three-layer architecture is good, but the use of components could have been better.
B	Maintainability is acceptable. Maintainable code has been emphasized and extensions have been planned for. The database layer is easy to extend, but in some places the code could have been better.
C	It is too costly to plan for maintainability, but the system should be suited for simple extensions.
D	Maintainability has been emphasized. Particular care was taken to ensure a good three-layer architecture, although some trade-offs were made to keep to the schedule.

5. Maintainability Assessment based on Structural Properties

The structural properties of the four systems were measured using an adapted version of the CK-metrics. Two different approaches were then used to assess maintainability of the systems. The detailed measurement procedure and the rationale between the two approaches are described in more detail in [5].
In the first approach, called *aggregation first*, all the measures were aggregated into summary statistics for the four systems. In the second approach, called *combination first*, the different measures for each class were combined and the classes were then categorized according to assumed maintainability. The tool used to extract the measures from the code was the M-System from Fraunhofer IESE.

The original CK set was adapted in the following way:

1. LOC (Lines of code), Comments (number of lines of comments) and Classes (Number of classes) were added.
2. The CBO measure has been shown to confound with size, and fan-out coupling has different effects than fan-in coupling. Hence, CBO was substituted with OMMIC (Call to methods in unrelated class) and OMMEC (Calls from methods in unrelated class).
3. When OMMIC was included. RFC was considered superfluous and removed.
4. The LCOM measure was substituted with TCC (Tight class cohesion), which is a normalized cohesion measure that has more discriminating power and is less influenced by size.

5.1 Aggregation first

Table 2 shows values for LOC, Comments and Classes and mean values and standard deviation for the other measures (the format is mean value/std).

System C has relatively high values and large standard deviations for size (WMC) and coupling (OMMIC, OMMEC) of classes, which indicates large and complex classes and an uneven design. Furthermore, System C has zero value for the inheritance measures, so inheritance was not used. The

Table 2. Summary statistics of CK metrics

	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

cohesion value (TCC) is high, probably much due to the size of the classes. The measures in Table 2 therefore indicate that System C will be difficult to maintain.

System D has a low measure for size and complexity and coupling, and a low standard deviation, but relatively high measures of inheritance. Despite very low cohesion, this indicates that System D will be easy to maintain.

System A has large coupling values, a rather large standard deviation for export coupling (OMMEC) and relatively low inheritance depth and high cohesion, while System B has good mean values. Due to lower coupling measures, System B was assessed as being more maintainable than System A.

5.2 Combination first

The second approach was to combine the different measures for each class and subsequently categorize each class as having low, acceptable, high, or very high values. The limits of each category were calculated from the 0 to 50 percentile, 50 to 75 percentile, 75 to 95 percentile, and above 95 percentile of the concatenation of all classes. The comparison criterion used was "The weighted sum of the criteria supporting the classification should be larger than the weighted sum opposing the classification". Table 3 shows the number of classes in each category for each of the systems.

Table 3. Number of classes in each category

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

Systems A, C and D had few classes with high or very high values, although, in the case of System C there is a large percentage of such classes due to the low total number of classes. System B has many classes with high and very high values, but also very many with low values. Considering the percentage of classes with high values, the ranking of the systems with respect to likely future maintenance effort is: D, A, B, C (in order of increasing effort).

These measures may give some indications of which classes, and how many, are likely to be difficult to maintain, but it is difficult to assess the consequences of such classes on the maintainability of complete software systems. In this case, we had the opportunity to compare four systems and can therefore rank the systems according to assumed maintainability and draw the tentative conclusion that the system that exhibits the worst values will create maintainability problems. In the typical situation, in which there is only one system to assess, interpreting the values will be much more difficult.

6. Expert Assessments

The expert assessments were conducted individually by two very experienced Java consultants[1] who did not know the results of the assessments based on structural measures. The first expert had 20 years experience of software development, including 10 years with Java development. The second had 10 years development experience, including six years with Java. Both were paid their normal consultancy fee for the work on the assessment and both delivered a report as the result. They assessed the code from the perspective of maintainers who are experienced Java programmers, but not familiar with details of the system. Due to there being few previous studies on expert assessments of Java software, it was decided to let the experts choose their own evaluation criteria on the basis of their experience with software development. For the same reason, it was decided not to ask the experts to quantify the assessment of each factor at this stage because the goal of the study was to identify a set of factors affecting maintainability and to obtain an overall assessment of the maintainability of each of the systems.

[1] In the study described in [5] the results were also validated by two expert assessments. One of them did not have industrial experience with Java development, so in this study he was substituted by an expert with long industrial experience of Java development.

Although the two experts did not communicate in any way, their criteria and conclusions were very similar. Due to the relative simplicity of the four systems, the experts were also asked to extrapolate beyond the system and attempt to predict the consequences of design decisions that they considered important for maintainability also of larger and more complex systems. This section describes the factors that were considered important by the experts and their assessments of the four systems according to these factors.

6.1 Factors Affecting Maintainability and their Assessment

An overview of the assessment is given in Table 4, and the opinions of the experts on each factor and its effect on maintainability is summarized below.

Choice of Classes and Names - The requirements of a software system usually indicate a number of obvious objects. In what follows these are denoted primary objects, which should be implemented as classes. These classes should be easy to identify to facilitate the mapping from domain and requirements to code.

Design - The design, including the use of design patterns, must be adapted to the actual project. The use of design patterns may make maintenance easier, because such patterns represent well-known solutions to commonly occurring problems in software development. However, 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.

The comprehensibility and, hence, the utility of individual design patterns has been shown to depend on the competencies of developers [18]. Therefore, a good initial design may not become a good implementation if the developers are not sufficiently competent. In fact, a good, but complex design may cause more harm than a simple and easy to understand design, because complex designs are more vulnerable to bad implementation practices.

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.

Components - Classes should be organized according to functionality or according to the layer of the code on which they operate.

Encapsulation - In Java there are three ways of ensuring encapsulation: 1) using public attributes, constants and methods, 2) using interfaces, and 3) using inner classes. Using public attributes, constants and methods is the simplest and most common way, but if not used consciously, public methods may require a specific sequence of method calls or may require an object structure to be established before a method is called. Such dependencies may create maintenance problems. There are also some potential pitfalls with the use of interfaces; if an interface implements several classes, it has the same effect as multiple inheritances, which may lead to confusion and lower maintainability.

In Java, inner classes may be used to hold internal structures that are not needed outside a specific class. Since Java methods return only one object, developers often create many small container classes where output from methods can be delivered. Using inner classes allows return values from methods to contain more than one variable. This reduces the use of these small classes and simplifies the design, making it more maintainable.

An additional aspect of encapsulation is that texts that may have to be changed should be collected as static variables and referenced elsewhere.

Inheritance - The impact of inheritance on maintainability depends on how it is used. Due to the small size of the systems, inheritance was not used extensively; there is typically only one level of inheritance. The maintainability in this case was assumed to depend on the distribution of functionality between the base class and subclasses. The base class should include enough “standard” functionality for example exception handling so that this must not be handled in each subclass. When this is satisfied, the class should be open to extensions, but closed to modifications. However, especially in the first iteration of a system, there is a balance between adding functionality to a base class or extending it, because it may not be obvious what generic functionality will be required by all or most subclasses. Furthermore, the use of inheritance increases the total number of classes, which may in itself decrease maintainability and should therefore be used with care.

Libraries – In this case, the use of libraries was very much related to the use of inheritance. Class libraries may allow developers to be more efficient because they provide good basic operations. However, the use of libraries may mean a greater amount of code, which in itself is less maintainable. The use of proprietary libraries may mean lower maintainability, because new developers will need to familiarize themselves with them. Proprietary libraries may also be influenced by the coding style of the developers that created the library, something that may make the code

Table 4. Expert Assessments

	A	B	C	D
Choice of Classes	Primary objects are implemented with classes that contain both data and logic.	Primary objects are implemented as containers. There are also additional, unnecessary containers for these objects.	Primary objects are mostly implemented as containers and most of the logic is separate from these.	Primary objects are implemented as containers. Each has a corresponding class for communication with the database.
Design	A good design with good use of design patterns.	A textbook example of a good design, but too comprehensive for this project. Unnecessary use of design patterns.	A simple design centred around two classes. Other classes are either containers or very small. No use of design patterns.	A comprehensive solution, with good use of design patterns.
Architecture	A three- layer architecture, but code is used across layers in some places.	A three-layer architecture, but the business layer is not completely de-coupled from the presentation layer.	No layered architecture, for example, just one large Java class for database management.	A three-layer architecture, but the business layer mostly contains commands to the database layer.
Components	Good use of components in the database layer, but not so good in the business layer. Some components are trivial.	Uses components, but not always successfully.	No use of components.	Mostly good use of components. Some of the components have very little content.
Encapsulation	Mostly good use of public methods, but too many methods are declared as public. Does not use interfaces or inner classes.	Good use of public methods and uses interfaces. However, many classes implement several interfaces. Good use of inner classes.	Does not use public methods very cons-ciously. Many large classes means that most variables and methods are used in many places. Does not use interfaces or inner classes.	Good use of public methods. Uses interfaces, but many of the interfaces do not have methods.
Inheritance	Mostly successful use of inheritance, but in some cases the base class does not contain all the functionality that is expected in a base class.	Too extensive use of inheritance. Confusions regarding whether functionality should be in the base class or the sub class.	No use of inheritance.	Mostly successful use of inheritance, but in many cases the base class does not contain all the functionality that is expected in a base class.
Simplicity	Unused code. Many almost identical methods for checking user input.	Many almost empty classes.	Classes with several functionalities, repeti-tions, inconsistencies in the use of SQL queries.	Many small classes with very limited responsibilities, some repetition in the code.
Naming	Some method names are very long.	Class names are often single, generic words.	Class names are mostly single, generic words.	Class names are often user functions.
Comments	Mostly good, but some redundant comments due to removed code.	Some classes lack overall comments.	Few overall comments on the classes. Most methods have good comments. Some comments are trivial.	Mostly good, but some comments are trivial and some are missing.
Libraries	Uses only standard Java libraries.	Uses a comprehensive proprietary library.	Uses only standard Java libraries.	Uses a proprietary library.
Technical platform	Standard tools	Uses some non-standard components.	Standard tools	Uses some non-standard components.

difficult for other developers to understand. The growth of standard Java libraries has made proprietary code libraries less important.

If a system has special needs that the library does not support, developers may have to alternate between services found in the library and making specialized code, something that may result in code that is more difficult to understand.

Simplicity – The code should not include statements that are very similar to each other. The code smell "DuplicateCode" is, perhaps, the worst [9]. The presence of several classes that are almost empty is another sign of code that may possess low maintainability, because it takes longer to identify a specific class when there are many classes (there is also a code smell for this situation, "LazyClass", a class that is not doing enough). Another factor related to simplicity is code that is commented out. This may be an asset because it can be included later; but it adds to size and complexity. System A had some unused code and Systems B and D had some small classes, but System C had long if-else statements for retrieving information and had statements that were almost identical.

Naming - The use of standard naming conventions for packages, classes, methods and variables eases understanding. In addition, the developers should use names to create a consistent schema that allows the reader to understand the relationship between methods and classes. All the companies followed mostly standard Java naming conventions, the only exception being D's use of class names.

Comments – The amount of comments was measured automatically and their quality was assessed by the experts.

Technical platform – An important part of systems maintenance is the ability to adapt to different environments, and many problems with systems maintenance are related to undocumented, implicit requirements that surface when a system is moved to a different environment. The use of a standard platform of tools simplifies this. A standard platform is, by definition, widely used; hence, it will be known by developers and will be supported by other companies. By contrast, the use of non-standard third party components poses a number of challenges related to using the components in further development. The developers have to put extra effort into understanding how to use the component. They will also have to understand how to replace the component in the future because such components may not be maintained or may become unavailable.

According to the expert assessments, 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.

The two experts agreed in their assessment of the individual factors for each of the systems. However, they disagreed slightly on the overall ranking of the systems. Expert 1 ranked the systems in the order A, D, C and B, while Expert 2 ranked Systems A and D together in first place and then System B before System C. The difference was probably 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.

6.2 Comparison of Assessments

We see from Tables 2, 3 and 4 that the expert assessments and the assessments based on structural measures gave quite similar results when used to rank the systems.

On the basis of structural measures, System D was ranked as the most maintainable and System C as the least maintainable system. Systems A and B had very similar measurement values, and in the aggregation-first approach that considered mean and standard deviation of the values for the different CK-metrics, System B was ranked as more maintainable than System A because of lower (better) values on the coupling measures. In the combination-first approach, System A was ranked before System B because System B had a higher number of classes that, overall, had high or very high measurement values. These assessments were in agreement with the opinions of the teams themselves.

The experts ranked the assumed maintainability of the systems slightly differently. They ranked System A as the most maintainable, with System D as a close second. They assessed both System B and System C as being potentially difficult to maintain, citing as reasons that System C did not follow basic object-oriented principles, while System B was too complex.

Many of the factors that the experts considered important were also measured implicitly using the structural properties. In addition, good measurement values on the structural attributes may, in general, be an indication of well-qualified developers who have maintainability in mind. Therefore, systems with good measurement values are also likely to be good with respect to other design factors (although this was not seen to be the situation in this case, where relatively good structural measures for System B "hid" potential maintainability problems).

The individual CK-metric in Table 2 that corresponds best with the expert assessment is the size measure WMC.

The factors that the experts considered important for maintainability in this context for small to medium systems that were developed using Java but that were not captured by the structural measurements were as follows:

- Choice of classes and names.
- Unnecessary classes, for example in the form of too many interfaces per class. A high percentage of small classes may also be an indication of such a problem.
- Division of classes into components.
- The distribution of functionality between the base class and subclasses. A high percentage of small classes may be an indication of too little functionality in the base class.
- Encapsulation and use of public methods.
- A design that is appropriate for the complexity of the system. In general, bad measurement values may be a sign of a design that is either too simple or too complex for the system, although this was not the case in the study.
- A good architecture where the presentation layer, business layer, and database layer are well separated.

7. Scope of Results

The scope of the results is Java systems. The structural properties were assessed using the CK-metrics. It is possible that other metrics may be more suited to this type of system. For example, one factor that was considered important for maintainability, but not detected using the CK-metrics, is the appropriate use of public methods. The MOOD set of metrics contains a measure for the use of such methods[10].

A threat to the validity of the results is posed by the assessments made the experts. They were both very experienced and their assessments were in accord. However, the assessments were subjective and different experts may consider other factors to be important and/or may assess Java systems differently according to the same factors.

The assessments based on the CK-metrics also relied on expert opinion to give an overall opinion on the maintainability of the systems and to rank the systems. There is no established way of combining the CK-metrics into one overall measure. Therefore, the results of these assessments may also be affected by subjective opinion. This represents a threat to the validity of the results, although probably to a much lesser extent than that posed by the expert assessments.

8. Conclusion and Future Work

It is important to be able to assess the maintainability of complete software systems, particularly for software clients. However, most research on the maintainability of software has focused on class-level measures and methods for identifying the least maintainable classes of individual software systems.

Related research in the field of software estimation indicates that the maintainability of software systems can best be assessed using a combination of expert assessments and methods based on structural measures of the code.

This paper has described the assessment of four functionally equivalent systems developed using Java. The systems were first assessed using structural measures and expert assessments. The results of the two assessments were mostly in agreement, but also point to some important factors related to maintainability that are not easily detected by structural measures. The results should therefore represent one (small) step in the direction of formulating a strategy for assessing the maintainability of complete software systems.

The contributions of this study are twofold: 1) it provides insights into the factors that experienced Java developers consider to be important for maintainability, and 2) it describes and exemplifies strengths and weaknesses of maintainability assessments that are based on structural measures and expert assessments.

The lessons learned from this study, from the point of view of software practitioners and software clients in particular, are that structural measures may be useful in assessing maintainability of software systems, but such measures should be used with care and should be combined with assessments of the factors described in Section 6.1.

Future work is planned to study agreement among experts when assessing maintainability of Java code, and also to expand the set of structural measures used in the assessment of the systems. Furthermore, the consequences for actual maintainability of the different designs used in these four systems will be investigated. More work is also needed on how to best combine expert-based and method-based assessments of maintainability.

Acknowledgements

I thank Hans Christian Benestad and Erik Arisholm for assessing the code using class-level measures, Per Einar Arnstad and Sindre Mehus for their assessments of the Java code, and Dag Sjøberg for obtaining the necessary funding for, and for organizing, the research project in which the four software systems were developed.

9. References

[1] AFOTEC Software maintainability evaluation guide. Department of the Air Force, HQ Air Force Operational Test and Evaluation Center, 1996.

[2] Anda, B., Benestad, H.C. and Hove, S.E. A Multiple-Case Study of Effort Estimation based on Use Case Points, in Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE'2005). IEEE Computer Society, Noosa, Australia, November 17-18, pp. 407–416, 2005.

[3] Arisholm, E. and Sjøberg, D.I.K. Evaluating the Effect of Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software. *IEEE Transactions on Software Engineering,* 30(8):521-534, 2004.

[4] Bakota, T., Ferenc, R., Gyimóthy, T., Riva, C. and Xu, J. Towards Portable Metrics-based Models for Software Maintenance Problems, in Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06), pp. 483 – 486, 2006.

[5] Benestad, H.C., Anda, B. and Arisholm, E. Assessing Software Product Maintainability Based on Class-Level Structural Measures, in Proceedings of the 7th International Conference on Product-focused Software Process Improvement (PROFES), edited by Jürgen Münch. Springer-Verlag, pp. 94-111, 2006.

[6] Briand, L. and Wuest, J. Empirical Studies of Quality Models in Object-Oriented Systems, *Advances in Computers*, Vol. 56, pp. 97-166, 2002.

[7] Chidamber, S.R. and Kemerer, C.F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering,* 20(6): 476-493, 1994.

[8] Darcy, D. and Kemerer, C.F. OO Metrics in Practice. *IEEE Software*, 22(6): 17-19, 2005.

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

[10] Harrison, R., Counsell, S.J., and Nithi, R.V. An Evaluation of the MOOD Set of Object-Oriented Software Metrics, *IEEE Transactions on Software Engineering,* 24(6): 491-496, 1998.

[11] IEEE standard for software maintenance. The Institute of Electrical and Electronics Engineers, Inc, New York, 1998.

[12] Jørgensen, M. Estimation of Software Development Work Effort: Evidence on Expert Judgement and Formal Models, Accepted for publication in the *International Journal of Forecasting*, 2007.

[13] Kirsopp, C., Shepperd, M. and Webster, S. An empirical study into the use of measurement to support OO design evaluation. In Proceedings of the 6th International Symposium on Software Metrics, Boca Raton, USA, November 4-6, pp. 230-241, 1999.

[14] Mayrand, J. and Coallier, F. System Acquisition Based on Software Product Assessment, in Proceedings of the 18th International Conference on Software Engineering (ICSE'96), pp. 210-219, 1996.

[15] Mäntylä, M.V. and Lassenius, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering,* 11(3):395-431, 2006.

[16] Schneiderman, B. *Software psychology: human factors in computer and information systems*. Winthrop, Cambridge, Massachusetts, 1980.

[17] Shepperd, M.J. System architecture metrics for controlling software maintainability. In Proceedings of the IEE Colloqium on Software Metrics, April 1-3, 1990.

[18] Vokač, M., Tichy, W., Sjøberg, D.I.K, Arisholm, E. and Aldrin, M. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment. *Empirical Software Engineering*, 9(3): 149-195, 2004.