Forming Consensus on Testability in Software Developing Organizations

Frans Mårtensson, Håkan Grahn, and Michael Mattsson
Department of Systems and Software Engineering
School of Engineering, Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{Frans.Martensson, Hakan.Grahn, Michael.Mattsson}@bth.se, http://www.bth.se/besq

ABSTRACT

Testing is an important activity in all software development projects and organizations. Therefore, it is important that all parts of the organization have the same view on testing and testability of software components and systems. In this paper we study the view on testability by software engineers, software testers, and managers, using a questionnaire followed by interviews. The questionnaire also contained a set of software metrics that the respondents grade based on their expected impact on testability. We find, in general, that there is a high consensus within each group of people on their view on testability. Further, we have identified that the software engineers and the testers mostly have the same view, but that their respective views differ on how much the coupling between modules and the number of parameters to a module impact the testability. Base on the grading of the software metrics we conclude that size and complexity metrics could be complemented with more specific metrics related to memory management operations.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging], D.2.8 [Metrics]

General Terms

Management, Measurement, Design, Human Factors.

Keywords

Testability, Software testing, Software metrics.

1. INTRODUCTION

In software developing organizations there exist a number of roles. These roles range from, e.g., project managers through software and hardware engineers to test engineers. As an organization grows and evolves, new people are introduced to the different roles. Each person brings their own knowledge and experience to the organization based on their background and education. Their background thus influences how they practice their role. As a result, an organization that from the beginning had a shared set of definitions and understandings between people in different roles, can after some time end up in a state where this is no longer the case. Roles can start to have different meanings of the same concept. But, when people in different roles no longer understand what a concept means to another role in the organization, it can become a source of misunderstandings, and also generate additional costs.

For example, an organization decides that the software system that they are developing needs to be improved, and a set of desirable quality attributes and requirements is selected. The changes made to a software system can be driven by many sources, ranging from business goals, e.g., new functionality requested by the customers, to pure maintenance changes, e.g., error corrections. In addition, the developers designing and implementing the system also have an impact on how the system changes. Therefore, it is important that all roles in the organization have an understanding of what the quality attributes mean to the other roles. One important quality attribute for a software system is testability [6, 8, 9], having high testability simplifies the task of validating the system both during and development and maintenance [14].

In this paper we examine a software developing organization. We look for different definitions of and views on testability between different roles in the organization. We devise a questionnaire and use it to gather the opinions and views on testability of the people in the organization. The questionnaire is then followed up with some additional questions raised during the analysis of the responses. The follow-up questions were posed during telephone interviews. Finaly we analyze and discuss the results of the examination. We plan a workshop where we will try to identify the reasons for the different opinions of the respondents, other than the ones that we have identified from the questionnaire.

The rest of the paper is organized as follows. In the next section we introduce software testing and testability. Then, in Section 3, we define the basis for our case study, e.g., the goals and participants in the study. In Section 4 and Section 5, we present the results from our questionnaire along with an analysis of them. Then, in Section 6 we discuss the validity of our findings. Finally, we conclude our study in Section 7.

2. SOFTWARE TESTING AND TESTABILITY

Software testing is the activity of verifying the correctness of a software system. The goal is to identify defects that are present in the software so that they can be corrected. Several classes of tests exist and the tests can be performed on a number of levels in the system. Typical tests used in software development are unit tests, regression tests, integration test, and system tests.

Different types of tests can be introduced at different points in the software life cycle. Unit tests are often introduced during the implementation phase, and focus on testing a specific method or class in the software system. Unit tests can be constructed by the developer as he writes the program code and used as a verification that the code that has been produced meets the requirements (both functional and non-functional) posed by the requirements specification.

Tests performed late in the development cycle are, e.g., integration tests and system function tests. The integration

tests test that the software modules that have been developed independently of each other still work as specified when they are integrated into the final system configuration. Integration testing becomes particularly important when the development organization is geographically dispersed, or when parts of the system have been developed with only little interaction between different development teams. System tests verify that all the integrated modules provide correct functionality, i.e., correct according to the requirements specification. System tests view the system from the user's point of view and look at the complete system as a black box which the user interacts with using some sort of user interface.

Catching defects early in the development process is an important goal for the development organization. The sooner a defect is identified the sooner it can be corrected. Software defects can be at the code level, in algorithms, but also at the design or architecture level. Defects at the design and architecture levels become more expensive to correct the later in the development cycle that they are identified, since larger parts of the design have been implemented. Hence the earlier the testing can begin, the more likely it will be that the completed software will be correct. Several development methods exist that put emphasis on testing, e.g., agile methods.

During software development different roles focus on different types of tests, depending on the organization. It is common that the programmers create and implement the unit tests while they implement their part of the system. Later tests, such as integration and particularly system tests, are usually done by a dedicated test team that has software testing as their main task.

One way of simplifying the repetition of tests is to automate the process. This is useful for, e.g., regression, unit, and performance tests. Automated testing can be implemented as a part of a daily build system. Test cases that have been automated can be executed once a build cycle is completed. Report generation and comparison to previous test results can be created as feedback to the developers. An example of such a system is Tinderbox by the Mozilla Foundation [13]. During the daily build it is also possible to collect data regarding source code metrics, e.g., the Maintainability Index [11] can be computed.

A software system that makes the testing activity easy to perform is described as having a high testability. Testability is a quality attribute [3] of a software system. It describes how much effort that is required to verify the functionality or correctness of a system, or a part of a system. One aspect of testability is to give the software testers useful feedback when something goes wrong during execution. Testability exists at a number of levels ranging from methods/classes through subsystems and components up to the system function level. Several definitions of testability exist, e.g., [6, 8, 9].

"Attributes of software that bear on the effort needed to validate the software product." [6]

"Testability is a measure of how easily a piece of hardware can be tested to insure it performs its intended function." [8]

"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." [9]

An important observation is stated in [6]:

"Testability is predicted on the definition employed, and there are numerous definitions. All definitions are legitimate; however, it is necessary to agree on terminology up front to avoid misunderstandings at test time."

The statement above leads us to the first focus in this study, i.e., how to identify different definitions of testability between roles in an organization. The second focus in our study is to identify those metrics the software developers and testers believe have an impact on testability. We focus our investigation on static analysis of a software system. Dynamic analysis requires an executable system, and that is out of scope of this study. Static analysis typically collects metrics by analyzing the complexity of the source code, i.e. counting the number of statements in and looking at the structure of the code [10]. There are metrics that can be collected at the method/algorithm level but there are also a number of object-oriented metrics that can be collected at the design level, e.g., inheritance depth, number of methods in a class, and coupling [7].

3. OBJECTIVES AND METHODOLOGY

Our case study was performed at a company that develops embedded control and guidance systems for automated guided vehicles. The company has worked with software development for more than 15 years, and has a development organization consisting of about 20 software engineers, 5 project managers, 3 product managers, and 2 software test engineers.

The company expressed concerns that there were misunderstandings between organizational roles regarding software testability and software testing. As part of the BESQ [2] project we devised this study to evaluate how the people in different roles in the organization define testability. We look for disagreement between the roles as well as within the roles.

Three roles (groups) are included in this case study: software engineers, test engineers, and managers. The group of software engineers includes the programmers and designers that develop the software for the control system. The group of test engineers includes people that perform system function testing of the completed software system. Finally, the third group consists of the project managers and product managers. Project managers manage the groups of software and test engineers, and make sure that the software development is proceeding according to schedule. The product managers represent the customers' interest in the development process, and are responsible for forwarding the requirements from the customers during the development and maintenance of the software.

Together with representatives from the company we determined two short-term objectives as well as one long-term objective. The two short-term objectives were:

1. To see how testability is defined by different roles in the organization.
2. To identify a set of software metrics that the developers indicate as useful when determining the testability of source code.

The third, long-term, objective will only be discussed briefly in this case study. We will not try to implement any changes in the organization yet.

3. Based on the results from objective 1, try to create a unified view of the testability concept within the organization.

To fulfill the first objective we need to determine the current state of the organization. We use a questionnaire to gather the

required information, which also enables us to analyze the results relatively easy. The questionnaire contains a set of statements regarding testability, and the respondents indicate to what extent they agree or disagree with each statement.

The second objective require a number of software engineers to grade a set of source code metrics. The idea is to identify which source code metrics that the respondents believe have an impact on the testability of a system. This information is also collected through the questionnaire. For each metric, the respondent indicates whether he/she believes the metric has a positive or negative impact on testability, and how large the impact would be. This part of the questionnaire is divided into two sub parts; one that focuses on metrics that can be collected from C/C++ code, and one that focuses on metrics that only are relevant for C++ code, i.e., object-oriented structural metrics [1, 4].

The third objective will be addressed through a workshop where the results of the survey will be presented to participants from the different roles. The intention is to discuss with them the different testability definitions that exist as well as the other roles' views on testability and expectations on their own role.

The questionnaire addressing the first two objectives contains 65 questions, which are divided into three categories. Category one contains questions related to the role of the respondent in the organization as well as background and previous experience. The second category contains questions regarding how the respondent view testability. This is formulated as a number of statements that the respondent agree or disagree with. The statements are graded using a five point Likert scale [12]. The scale ranges from 1 to 5, where 1 is that the respondent does not agree at all, 3 indicates that the respondent is neutral to the statement, and 5 indicates that the respondent strongly agrees with the statement. The third category contains a number of source code metrics that is used to assess the testability of C/C++ source code. These metrics are also graded using a five point Likert scale. For each question we also ask how sure the respondent is on the answer. These questions are also graded using a Likert scale.

The questionnaire was distributed as an Excel file via e-mail and the respondents sent their replies back via e-mail. Together with the Excel file we sent a short dictionary with descriptions of concepts and definitions that were used in the questionnaire. This was done in order to minimize the amount of questions that the respondents might have and to make sure that all respondents used similar definitions of basic concepts. We sent the questionnaire to all people with the roles that we wanted to examine.

4. RESULTS AND ANALYSIS OF TESTABILITY STATEMENTS

We distributed 25 questionnaires and got 14 responses, resulting in a 56% response rate. The number of responses for each of the groups is found in Table 1. The number of responses was to few to apply statistical methods. Instead, we rely on quantitative and qualitative reasoning based on the answers. The first part of the questionnaire (general testability statements) was answered by all 14 respondents, and the second part (C/C++ metrics) was answered only by the software engineers.

Table 1. The number of replies divided per role.

Role	Nr of replies
Software Engineer	9
Test Engineer	2
Project/Product Manager	3
Total number of replies	14

The statements in the first part of the questionnaire are listed in Table 2, and are all related to definitions of testability. The statements can be grouped into three groups: S1, S3, S5, S6 and S12 are related to the size and design of a software module, S2, S4, and S7 are related to the size of a software module only, and S8, S9, S10, and S11 are related to the internal state of a module.

Table 2. Statements regarding testability.

ID	Statement
S1	Modules with low coupling have high testability.
S2	Functions with few parameters have high testability.
S3	Modules with high coupling have low testability.
S4	Modules with few lines of code have high testability.
S5	Modules with few public methods in their interface have high testability.
S6	Modules with many public methods in their interface have high testability.
S7	Functions with many parameters have high testability.
S8	If it is easy to select input to a module so that all execution paths in it are executed, then it has high testability.
S9	If it is likely that defects in the code will be detected during testing then the module has high testability.
S10	If it is easy to set the internal state of a module during testing then it has high testability. Modules that can be set to a specific state makes it easier to retest situations where faults have occurred.
S11	If it is easy to see the internal state of a module then it has high testability. Modules that for example log information that is useful to the testers/developers have high testability.
S12	If a module has high cohesion, then it has high testability.

The analysis is done in several steps. First, we compare the responses by the people within each group (role) in order to evaluate the agreement within the group. Second, we compare the answers between the groups in order to evaluate agreement or disagreement between the groups. Third, we compare the expectations the software engineers and test engineers have on each other regarding testability.

4.1 Consistencies Within the Roles

The first analysis of the responses is to look for inconsistencies within the roles. We want to see if all respondents in a role give similar answers, and if they differ, where the inconsistencies are. The answers are summarized so the

frequency of each response is identified, and we also evaluate the distribution of responses. The responses for each question is grouped into four categories for further analysis. The categories are defined as:

1. **Consensus**. There is a clear consensus between the respondents.
2. **Consensus tendency**. There is disagreement but with a tendency towards one side or the other.
3. **Polarization**. There are two distinct groupings in the responses.
4. **Disagreement**. There is a clear disagreement between the respondents.

If a statement is placed in the Consensus and Consensus tendency categories, it is good, i.e., the respondents agree with each other and a tendency can be seen even though the responses might spread somewhat. The Polarization category is less good and indicates that there exists two different views on the statement among the respondents. The Disagreement category is also less good since there are several views of the statement and no real consensus among the respondents exists.

4.1.1 Software Engineers

The software engineers' responses can be found in Table 3. The responses to statements S1, S3, S5, and S10 belong to the Consensus category since there are a clear consensus in the answers. Statements S2, S6, S7, S9, and S11 we put in the Consensus tendency category where we have a tendency towards agreement or disagreement with the statement. In the Polarization category we put statements S8 and S12 since there are two groupings in the answers. Finally, in the Disagreement category we put statement S4.

Table 3. Summary of replies from the software engineers.

Answer:	1	2	3	4	5	Result Category
S1	0	0	0	2	6	Consensus
S2	0	1	2	2	3	Consensus tendency
S3	0	0	1	3	4	Consensus
S4	0	2	2	1	3	Disagreement
S5	0	1	4	2	1	Consensus
S6	2	2	3	1	0	Consensus tendency
S7	3	2	2	1	0	Consensus tendency
S8	0	0	3	0	5	Polarization
S9	0	0	3	2	3	Consensus tendency
S10	0	0	1	4	3	Consensus
S11	0	1	2	2	3	Consensus tendency
S12	0	0	3	1	4	Polarization

The answers show that there are good consensus among the software engineers regarding coupling of modules, the number of public methods, the number of function parameters, and their relation to testability. There is also a high degree of consensus that modules have high testability if defects will be detected during testing, and if it easy to set and view the internal state of the module.

The software engineers have different opinions about high cohesion and the easiness of selecting input data for testing all execution paths. These issues are subjects for further investigation and discussion in the organization. Finally, there is a large disagreement if a module with few lines of code has high testability or not.

It is good that so many statements are agreed upon by the respondents. There are only two statements where the respondents form two distinct groupings, and only one statement where no clear consensus can be identified. This leads us to believe that the software engineers, as a group, have a rather coherent view on what testability is, although some disagreements exist.

4.1.2 Test Engineers

The responses from the test engineers can be found in Table 4. The number of respondents in this role is only two. Therefore, it is more difficult to place the statements in the four categories. In category Consensus we put the statements S1, S3, S5, S6, S8, S9, S10, and S12 since the respondents give similar responses to the statements. Statements S2, S4, S7, and S11 are placed in category Disagreement because the responses are further apart. The responses could not be placed in the Polarization category since they are spread between both agree and not agree to the statements. The issues where there exist disagreements are functions with few and many parameters, few lines of codes for a module, and viewing the internal state.

Table 4. Summary of replies from the test engineers.

Answer:	1	2	3	4	5	Result Category
S1	0	1	1	0	0	Consensus
S2	1	0	1	0	0	Disagreement
S3	0	2	0	0	0	Consensus
S4	0	1	0	0	1	Disagreement
S5	0	1	1	0	0	Consensus
S6	0	1	1	0	0	Consensus
S7	0	1	0	1	0	Disagreement
S8	0	0	0	1	1	Consensus
S9	0	0	0	2	0	Consensus
S10	0	0	0	1	1	Consensus
S11	0	1	0	0	1	Disagreement
S12	0	0	1	1	0	Consensus

4.1.3 Managers

The responses from the managers can be found in Table 5. They suffer from the same problem as the testers, i.e., we only got three responses to the questionnaire from this role which makes the categorization of the responses difficult. Statements S5 and S6 are placed in category Consensus, and statements S4 and S9 in the category Consensus tendency. The rest of the statements are placed in the Disagreement category since the respondents disagree on the grading of the statements. When one of the respondents agree then two disagree and vice versa. This group of the respondents is the one that has the most disagreement in their responses.

Table 5: Summary of replies from the managers.

Answer:	1	2	3	4	5	Result Category
S1	0	1	0	1	1	Disagreement
S2	0	2	0	1	0	Disagreement
S3	0	1	0	2	0	Disagreement
S4	0	1	1	1	0	Consensus tendency
S5	0	1	2	0	0	Consensus
S6	0	0	1	2	0	Consensus
S7	0	2	0	1	0	Disagreement
S8	0	1	0	1	1	Disagreement
S9	0	1	1	1	0	Consensus tendency
S10	0	1	0	2	0	Disagreement
S11	0	1	0	2	0	Disagreement
S12	0	1	0	2	0	Disagreement

From the managers viewpoint we see that the amount of public methods has no impact if a model has high testability or not (S5 and S6). They also believe that defects detected in code and modules with few lines of code indicate high testability. For all other statements there are differences in opinions, and this has to be addressed.

4.2 Differences Between Roles

The next analysis step is to compare the view on testability between the different groups (roles). In order to make the groups easier to compare, we aggregate the responses for the groups. The results of the aggregation can be found in Table 6. The numbers are translated into their literal meaning, i.e., 5 - Strongly agree, 4 - Agree, 3 - Neutral, 2 - Do not agree, and 1 - Strongly disagree.

Table 6. Aggregation of replies for the groups.

	Software engineers	Test engineers	Managers
S1	Strongly agree	Do not agree	Neutral
S2	Strongly agree	Do not agree	Neutral
S3	Strongly agree	Do not agree	Neutral
S4	Agree	Neutral	Neutral
S5	Neutral	Neutral	Neutral
S6	Neutral	Neutral	Neutral
S7	Do not agree	Do not agree	Neutral
S8	Agree	Strongly agree	Neutral
S9	Agree	Agree	Neutral
S10	Agree	Strongly agree	Neutral
S11	Agree	Agree	Neutral
S12	Strongly agree	Agree	Neutral

From the aggregation we find that the software engineers and the test engineers do not agree on three main statements (S1, S2, and S3). These are statements that are related to the design and size of software, i.e., coupling-cohesion vs. high-low testability and few function parameters. For the remainder of the statements there is mainly an agreement between the two roles.

Most of the managers' answers are neutral in the aggregation. The reason is that the responses from the managers had a large spread. One respondent answered similarly to the software engineers and another answered almost the direct opposite, and both were very sure on their answers. The differences in opinion can maybe be attributed to the backgrounds of the managers. One of them had never worked with testing, while the other one had long test experience. The differences also make it hard to make statements about the expectations on the other roles from the managers since they inside their group have different opinions, making the relations to the other groups hard to interpret. As mentioned earlier, this difference in opinions must be unified.

4.3 Understanding and Expectations Between Software Engineers and Test Engineers

We made follow-up interviews which focused on how the respondents perceive the expectations of the other roles that participated. The additional questions regarding this aspect complement and further focus the results of the study. The follow-up interviews were done over telephone. We did not include the managers since they as a group had a to scattered view on the statements. Hence, we discuss the expectations of the software engineers on the test engineers in Table 7, and vice versa in Table 8. For each statement the respondent answers what he/she thinks that the people in other role would answer. This give an indication of how much the roles are aware of each others opinion of testability.

Table 7. Software engineers' expected answers from the test engineers.

	Expected answers	Actual answer
S1	Strongly agree	Do not agree
S2	Agree	Do not agree
S3	Strongly agree	Do not agree
S4	Neutral	Neutral
S5	Neutral	Neutral
S6	Neutral	Neutral
S7	Do not agree	Do not agree
S8	Strongly agree	Strongly agree
S9	Strongly agree	Strongly agree
S10	Strongly agree	Strongly agree
S11	Strongly agree	Agree
S12	Agree	Agree

Overall we conclude that the software engineers and the test engineers seem to have a good understanding of each others interpretation of testability. The answers only differ on three statements, S1, S2, and S3, where both roles predicted different answers from the other role than they actually gave. From the interviews we think that the difference can be

attributed to different interpretations of the concepts of high and low coupling in object-oriented design.

Table 8. Test engineers' expected answers from the software engineers.

	Expected answers	Actual answer
S1	Strongly agree	Agree
S2	Neutral	Agree
S3	Do not agree	Strongly agree
S4	Strongly agree	Agree
S5	Neutral	Neutral
S6	Do not agree	Do not agree
S7	Do not agree	Do not agree
S8	Strongly agree	Agree
S9	Strongly agree	Agree
S10	Strongly agree	Agree
S11	Strongly agree	Agree
S12	Strongly agree	Strongly agree

5. SELECTION OF TESTABILITY METRICS

The second objective of our study is to identify and select a number of software metrics that can be collected and used to assess the testability of a system under development. The data for this selection are collected through the second part of the questionnaire. The statements and questions from this part of the questionnaire is presented in Table 9. Statements M1 to M9 are general questions related to code size. Statements M10 to M23 are questions related to C and C++, since those are the major programming languages used at the company. Finally, statements M24 to M38 are related only to C++. The C++ related metrics cover aspects such as class structures and inheritance while the C/C++ metrics focus on metrics related to code structure, methods, and statements. The participants in the study answered both in what way a statement impacts testability (improve or deteriorate) as well as how much it impacts testability relative to the other metrics.

We got 5 responses where all questions were answered, and 4 responses where only questions M1 to M23 were answered. The reason given by the respondents for not answering all questions was usually that they felt unsure about the C++ statements since they usually worked with the C programming language.

The results are analyzed in a similar way as in the previous sections. We aggregate the results and translate them from numbers to their literal meaning in order to make the results easier to compare as well as more readable. The mapping for the impact is done as follows: 1 - Very small, 2 - Small, 3 - Average, 4 - Large, 5 - Very large, and for the direction: 1 - Very negative, 2 - Negative, 3 - No impact, 4 - Positive, 5 - Very positive. We focus our analysis on the metrics that the respondents identify as the ones with large positive or negative impact on the testability of a system. The results of the aggregation is presented in Table 10 (large negative impact) and Table 11 (large positive impact).

Table 9. Statements regarding how different code metrics impact testability.

ID	Metric Statement
M1	How many lines of code the module contains.
M2	How many lines of code that each function contains.
M3	How many lines of comments the module contains.
M4	How many lines of comments that each function contains.
M5	How many parameters that are passed to a function.
M6	The length of the function name.
M7	The length of the class name.
M8	The number of lines of comments that are present directly before a function declaration.
M9	The number of lines of comments present inside a function.
M10	The number of variables present in a struct.
M11	The size of macros used in a module.
M12	The number of macros used in a module.
M13	The number of functions in a module.
M14	The number of parameters of a function.
M15	The number of macros called from a module.
M16	The number of times that a struct is used in a module.
M17	The number of control statements (case, if, then, etc.) in the module.
M18	The number of method calls that is generated by a method.
M19	The number of assignments that are made in a module.
M20	The number of arithmetic operations in a module.
M21	The presence of pointer arithmetic.
M22	The number of dynamic allocations on the heap.
M23	The number of dynamic allocations on the stack.
M24	The total number of classes in the system.
M25	The number of classes in a module (name space?).
M26	The number of interfaces that a class implements.
M27	The number of classes that a class inherits.
M28	The number of classes that a class uses.
M29	The number of methods present in a class.
M30	The number of private methods present in a class.
M31	The number of public methods present in a class.
M32	The number of private variables in a class.
M33	The number of public variables in a class.
M34	The number of overloaded methods in a class.
M35	The number of calls to methods inherited from a superclass.
M36	The size of templates used in a module.
M37	The size of templates used in a module.
M38	The number of different template instantiations.

Of the metrics with negative impact, see Table 10, we find that the developers focus on memory management aspects of the source code as well as the structural aspects. This is interesting as there is usually little focus on what the source code actually does and more on how understandable code is through its structure and complexity [11]. This indicates that structural

measures need to be complemented with some measures of the occurrence of memory related operations in the source code. Traditional size measures such as the lines of code in files and methods are also present in M1, M2, and M5. Source code complexity measures such as cyclomatic complexity [10] can also be seen in M17. Finally, we find class structure and inheritance measures related to object-oriented metrics [1, 4] in M26 and M27.

Table 10. Metrics with large negative impact on testability.

ID	Metric	Impact	Direction
M1	How many lines of code the module contains.	Large	Negative
M2	How many lines of code that each function contains.	Large	Negative
M5	How many parameters that are passed to a function.	Large	Negative
M17	The number of control statements (case, if, then, etc.) in the module.	Large	Negative
M21	The presence of pointer arithmetic.	Large	Negative
M22	The number of dynamic allocations on the heap.	Large	Negative
M23	The number of dynamic allocations on the stack.	Large	Negative
M26	The number of interfaces that a class implements.	Large	Negative
M27	The number of classes that a class inherits.	Large	Negative

The metrics that are graded as having a large positive impact on the testability of a system is presented in Table 11. These metrics can also be divided into two groups: the first is related to object-oriented structure (M24, M28, M30, M31, and M32) and the second is related to documentation, e.g., lines of comments (M3) and descriptive function names (M6).

Table 11. Metrics with large positive impact on testability.

ID	Metric	Impact	Direction
M3	How many lines of comments the module contains.	Large	Positive
M6	The length of the function name.	Large	Positive
M24	The total number of classes in the system.	Large	Positive
M28	The number of classes that a class uses.	Large	Positive
M30	The number of private methods present in a class.	Large	Positive
M31	The number of public methods present in a class.	Large	Positive
M32	The number of private variables in a class.	Large	Positive

Finally, we find three metrics that are graded as having a large impact but rated as neither positive nor negative, see Table 12. We interpret these results as an indecisiveness from the respondents, that the metric should have an impact on testability but that there is no feeling for towards which direction the impact is.

Table 12. Metrics that have an unclear impact on testability.

ID	Metric	Impact	Direction
M13	The number of functions in a module.	Large	No impact
M25	The number of classes in a module.	Large	No impact
M33	The number of public variables in a class.	Large	No impact

From the tables we can see that the common size and structure metrics are identified as properties that impact the testability of a system. But we also find properties related to memory management and addressing (M22, M23, and M21). It is common to focus the testability discussion on the understandability of the source code and not so much on what the source code actually does [7, 10, 11]. Our results indicate that memory management metrics should be taken into account when assessing the testability of software developed by this organization. We believe that this is data that relatively easy can be gathered from the source code.

Because of the developers' interest in memory related operations, we think that it would be interesting to complement existing testability measures, such as the testability index [7], with these metrics to see if the accuracy of the measure is improved. When creating such a measure it would also be possible to tune it to the developing organization and the systems they develop.

6. DISCUSSION

In this section we shortly discuss some validity aspects in our study. The first issue concerns the design of the questionnaire. It is always difficult to know whether the right questions have been posed. We believe that we at least have identified some important issues considering the overall view on testability by different roles in the company. The questionnaire was discussed with company representatives before it was issued to the software developers. In future studies, the questionnaire can be further refined in order to discern finer differences in peoples' opinion on testability. More statements would perhaps give a better result.

The second issue concerns the number of replies from the questionnaire, i.e., we only got 14 replies in total. The low number of replies prohibit us from using statistical methods for analysis. Instead, we have relied on qualitative reasoning and interviews to strengthen the confidence in the results. Further, the low number of respondents makes it difficult to generalize from the results. An interesting continuation would be to do a replicated study in another company.

The third validity issue is also related to the number of replies. There is a clear imbalance in the distribution of replies, most of the respondents were from one role, i.e., the software engineers. This results in a higher confidence in the results from one group than from the other groups. However, the difference in the number of responses per group reflects the

distribution of people working in the different roles at the company.

7. CONCLUSION

Software testing is a major activity during software development, constituting a significant portion of both the development time and project budget. Therefore, it is important that different people in the software development organization, e.g., software engineers and software testers, share similar definitions of concepts such as testability in order to avoid misunderstandings.

In this paper we present a case study of the view on testability in a software development company. We base our study on a questionnaire distributed to and answered by three groups of people: software engineers, test engineers, and managers. The questionnaire was then followed up by interviews.

Our results indicate that there is, in general, a large consensus on what testability means within each group. Comparing the view on testability by different groups, we find that the software engineers and the test engineers mostly have the same view. However, their respective views differ on how much the coupling between modules and the number of parameters to a module impact the testability.

We also evaluate the expected impact of different code metrics on testability. Our findings indicate that the developers think that traditional metrics such as lines of code and lines of comments as well as object-oriented structure and source code complexity are important. But we also found that the developers think that the presence of memory operations, e.g., memory allocation, has a negative impact on the testability of software modules. We think that common weighted metrics such as testability index can be complemented by the collection and integration of this information, depending on the type of domain that the organization is operating in.

Future work include organizing a workshop at the company where both software developers and software testers participate. The goal of the workshop will be to enhance the awareness of the different views on testability, based on the results presented in this paper, and also to reach some agreement on testability within the organization.

Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" http://www.bth.se/besq. We would like to thank Danaher Motion Särö AB [5] for their time answering our questions, as well as many interesting discussions and ideas.

REFERENCES

[1] B. Baudry, Y. Le Traon, and G. Sunyé, "Testability Analysis of a UML Class Diagram," *Proc. of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, pp. 54-63, June 2002.

[2] Blekinge - Engineering Software Qualities (BESQ), http://www.bth.se/besq/

[3] J. Bosch, *"Design & Use of Software Architectures,"* Pearson Education Limited, ISBN 0-201-67494-7.

[4] M. Bruntink and A. can Deursen, "Predicting Class Testability using Object-Oriented Metrics," *Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 136-145, September 2004.

[5] Danaher Motion Särö AB, http://www.danahermotion.se

[6] Encyclopedia of Software Engineering, 2nd ed., Wiley-Interscience, ISBN 0-471-37737-6, December 2001.

[7] V. Gupta, K. K. Aggarwal, and Y. Singh, "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability," *Journal of Computer Science* vol. 1, pp. 276-282, 2005.

[8] M. Hare and S. Sicola, "Testability and test architectures," *Proc. of the IEEE Region 5 Conference, 1988: 'Spanning the Peaks of Electrotechnology'*, pp. 161 - 166, March 1988.

[9] Institute of Electrical and Electronics Engineers, "*IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries,"* New York, NY, 1990.

[10] T.J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering SE-2,* pp. 308-320, 1976.

[11] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities," *Proc. of the 11th International Conference on Software Maintenance (ICSM'95)*, pp. 295-303, October 1995.

[12] C. Robson, "*Real World Research*," Blackwell publishers, ISBN 0-631-21305-8, 2002.

[13] Tinderbox, Mozilla foundation. http://www.mozilla.org/projects/tinderbox/, last checked 20051012

[14] J.M. Voas and K.W. Miller, "Software Testability: The New Verification," *IEEE Software*, 12(3):17-28, May 1995.