CHALMERS

An Approach to Improve Quality of Software Using Metrics and Technical Debt

A case study within Model-Driven Development Environment

Master of Science Thesis in the Programme Software Engineering and Technology

BJÖRN PETERSSON
SHENG ZHANG

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

An Approach to Improve Quality of Software Using Metrics and Technical Debt
A case study in Model-Driven Development Environment

BJÖRN PETERSSON
SHENG ZHANG

Examiner: MIROSLAW STARON
Supervisor: ROBERT FELDT

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

ACKNOWLEDGMENT

The authors would like to express appreciation and thanks to Volvo GTT for providing time and resources supporting the study. The authors are particularly thankful for the support and guidance given by Rolf Nilsson and Kristian Kinderlöv throughout the conducted study, as well as the important contribution by the development team at the studied Volvo unit. Furthermore, the authors are grateful of academic supervision and support from Professor Robert Feldt, Chalmers University of Technology.

Preface

This Master of Science Thesis is reported here in a hybrid format, i.e. the main content of the Work is reported as a scientific article conforming to the Empirical Software Engineering Journal's template, complemented by additional appendices relevant for the examination at Chalmers University of Technology.

Table of Contents

Received: date / Accepted: date

An Approach to Improve Quality of Software Using Metrics and Technical Debt

A case study within Model-Driven Development Environment

Björn Petersson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
bjorpe@student.chalmers.se

Sheng Zhang
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
shengz@student.chalmers.se

Abstract

BACKGROUND: As software products play vital roles in embedded systems, software quality has raised much attention in the software engineering field, especially those using model-driven development. However, software doesn't have physical features, so it is hard to measure and monitor its quality.

OBJECTIVE: To cope with the difficulty of measuring and monitoring software quality, a framework was developed and proposed to Volvo Group Truck Technology (GTT). After studies, analyses and discussions, software metrics and technical debt have been chosen and applied in this framework.

METHODS: The framework was developed based on research result from surveys, interviews, workshops and literature reviews. Software metrics were picked and applied in the framework to get basic measurements, the raw data was transferred, analyzed and presented with a modified form of technical debt so to fit this development team's requirements. Finally, the validity of the study was confirmed by another survey and historical data analysis. The framework is built to fit a model-driven development environment.

RESULTS: Team members - including developers, testers, architects and product managers - gave positive feedback to this framework after it was applied in the development environment. Furthermore, statistical analyses carried out on historical data supported the correctness of the framework.

CONCLUSIONS: Software metrics can help to analyze, measure and monitor software quality. Furthermore, potential risks could be reduced by improvements suggested in the report, such as i.e. splitting a method because it has too many lines of code or too high complexity. It can be found in this study that by combining software metrics and technical debt, the framework in the project proved to be an efficient support-tool to improve software quality. Furthermore, the framework has the potential to be adopted in other model-driven development teams and environments.

Key Words **- Code Quality, software metrics, Technical Debt, Model-Driven Development, Software Quality Improvement**

I. INTRODUCTION

In software systems not only functionality and the outward design are important, but also how it is coded, and designed architecturally with modules and connections between them. Software can function even if the quality of the written code and modules are low, though the software code may hide or introduce bugs when changed [50]. The quality of written code, modules and connection between modules are hereafter mentioned as Internal Software Quality (ISQ).

ISQ is connected with success of software projects and related costs; one of the reasons is that a project with good ISQ has reduced cost of maintenance [50, 51, 54]. However, there is no easy way to define measurements of software quality [3,4] and therefore also ISQ. As software systems increase in size as well as complexity, the importance of software quality rise and at the same time the software quality becomes harder to measure [5,6,50].

Studies also show that fixing defects in the late phases of a software development process is costly and can delay the whole development process [40]. That considerable cost can be saved if defects are prevented at an early stage is an accepted concept in the software engineering research field. Bad code-design can lead to defects and it does also increase the cost of maintenance, some reasons for this is reduced readability and complexity of the code, and thereby, system [7,53].

Model-Driven Development (MDD) is when development is based on models as a primary artifact and from which code, documentation and tests are derived [52]. Rational Rhapsody is a tool that automatically derives code based on models created by developers and which is the tool used at GTT. ISQ of MDD projects remain a neglected topic in the academic area. Only few papers can be found focusing on this, none of them offering detailed solutions to measure or improve ISQ for MDD projects. Moreover, none of the papers focused on using the combination of metrics [1] and Technical Debt (TD)[2].

The purpose of this study was to measure and monitor the ISQ and indirectly increase it in a system using Rational Rhapsody. This was done by gathering metrics on the state-charts and code written by the programmers. An important part of gathering metrics and measuring the ISQ was to pinpoint

special classes and methods, which were poorly designed and written, and that had a risk of containing bugs [53].

In the end of this study, a framework has been proposed for the team. The framework, which not only focused on ISQ of MDD project but also on visualizing the quality trends and monitoring them, it has also been evaluated with positive result. This approach included some detailed implementation and definition for code metrics as well as some general suggestions for the whole development process. The proposed framework was combined by tools which run analyzes of source code/models. The tools were also integrated with auto-compiling system to regularly generate a report based on TD, which is a term that is used to describe artifacts in code that will cost more and provide less quality in the long run [25]. This report is shown to managers and related personnel in the development team (in the following frequently referred to as the "team" or the "development team").

Since the topic ISQ is firmly connected to implementation of software code (models, in case of MDD), the main focus of this study was on the metrics and the ways to analyze and monitor them. Furthermore, the introduction of the concept of TD made it possible for developers and managers to communicate in a new way. The ISQ was not only reflected and visualized but also shown in the form of TD which is important for managers, especially for project managers.

In the earlier stages of the study research-questions with sub-questions were created to be answered:

RQ1. How to measure software quality in a MDD project?

SQ1.1 Which metrics are useful in a MDD project?

SQ1.2 How to apply metrics in a MDD project using Rational Rhapsody?

RQ2. How to improve software quality in a MDD project?

SQ2.1 Which approach is suitable for GTT's situation?

SQ2.2 How can results from metrics help with improving software quality?

SQ2.3 How to measure the improvement?

RQ3. How to present the result of ISQ improvement study?

SQ3.1 How to raise attention of ISQ for both developers and managers?

SQ3.2 Can technical debt works as a bridge between developers and managers when it comes to quality?

SQ3.3 How to present the result in a more understandable way?

II. Literature Review

The literature review consist of three topics; i) studies on metrics – fundament of the whole study; ii) ISQ improvement studies in practice; and iii) MDD quality researches and metric collecting.

A. Metric Studies

"Software metrics" is used as a name for many different areas, it varies from being used for measurements in software engineering to models predicting software quality or the amount of resources needed [9,10]. In this study, the term "software metrics" is referred to measurements, in numbers, of the software product at the development team in Volvo GTT.

Software metrics were used almost as early as the beginning of software engineering and some metrics from that time are still used. These early metrics from the late 1960's, Lines Of Code and other size-based metrics, are regarded both as successes and failures. The reason for them being a failure is that they were often misused [9] or lacked comprehensive definition [10]. There are many papers that discuss the usefulness of metrics, an example is Mills [1] that states that metrics can be a resource to increase software productivity and software quality [1]. Another example is Fenton and Neil [9] who write that its most significant role is to help in managerial decisions in software development [9].

There are also many papers that warn about risks related to using metrics. Often metrics are gathered in large quantities, and then never used, or needed. Sometime right metrics are gathered but never looked at because there are too many metrics and too much information to search through [10]. Judging by this information it is very important to have a goal/reason for integrating a metric into the system [1,10].

A very important part within the field of metrics is proper definitions, a metric without definition can be hard to understand and read. Westfall [10] compared metrics without definition with the speed of a car, to know the speed of a car the unit for speed (kilometers per hour/miles per hour) is required [10]. This means that there must be a proper definition to each metric to explain what the metric is measuring and how to read the metric. Mills [1] also mentions the importance of definitions when writing how a good metric should be: "simple, precisely definable - so that it is clear how the metric can be evaluated" [1].

Metrics that can be used in all projects and by all companies are hard to create, instead specific metrics can be constructed based on stakeholders wishes [1,10]. There exist metrics which are constructed and tested specifically for object-oriented environments. It also exists metrics that can be adopted into object-oriented environments [11,12]. Below is eight metrics which were chosen as the backbone for discussions in the study. The reason for them to be chosen was to test if object-oriented environment metrics could be used in a MDD environment and also because they have been tested in older studies [11,12]:

Cyclomatic Complexity (CC): Measures the complexity of a method, calculated by a summarization of all linearly independent paths through a software source code. CC should be below ten in a method, else the complexity is deemed to be too high [11].

Size: Evaluates how easily the code can be understood by developers and maintainers. Size can be measured in a variety of ways; some ways consist of counting physical Lines Of Code (LOC), number of statements and number of blank lines. Size thresholds differ depending on the code language used, but generally bigger size means the code will be harder to understand and this results in having less understandability, maintainability and reusability [11].

Comment Percentage (CP): CP is calculated by taking the number of comments, in any form, and dividing it by the number of physical lines of codes less the number of blank lines. Comments ease understandability, maintainability and reusability when they come in the right amount. Software Assurance Technology Center (SATC) has found that the most efficient comment percentage is when it is close to 30% [11].

Weighted Method per Class (WMC): The sum of the complexity of all methods in a class or sum of all methods in a class is the definition of WMC. The complexity of methods is calculated in the same way as CC. Measuring the complexity of the methods can be hard depending on whether the methods are accessible due to inheritance. The higher the methods the

more application specific the class becomes and it will also reduce the reusability of the class [11,12].

Coupling Between Object Classes (CBO): Classes are coupled when one class uses methods or instance variables defined in another class. CBO is the number of classes a class is coupled to. High coupling increases complexity, decreases reusability and a class with high coupling needs more rigorous testing [11,12].

Response For a Class (RFC): This metric looks at the complexity of a class by comparing the amount of methods it has combined with how much communication it has with other classes. The higher number of methods that can respond to a call the more complex the class becomes and by that testing and debugging will become more difficult [11,12].

Lack of Cohesion in Methods (LCOM): Cohesion in a class means that the methods in the class perform actions that are related. When there is no cohesion the code becomes more complex but may work as well as a class with high cohesion [11,12]. There exist different ways of calculating cohesion, the one that is explained below is a version proposed by Henderson-Sellers:

If m is the number of methods in the class, a is the number of attributes in the class, mA is the number methods that access the attribute a and sum(mA) is the sum of all mA over all the attributes in the class. The method to calculate LCOM is:

Formula 1: $LCOM = \frac{m - \frac{sum(mA)}{a}}{m-1}$

This method results in a number between zero and two and if the value is higher than one it should be seen as a warning and the class should be divided into subclasses [13].

Depth of Inheritance Tree (DIT): DIT is calculated for each class and from the number of ancestors that class has. The depth of the class is determined by how far it is from the top of the inheritance tree, the root, to the class node. The higher the number of ancestors a class has the more methods are used and the more complex the class becomes. With increasing inheritance the system will get a higher design complexity, but it also increases the chance of reuse of methods that is inherited [11,12].

Number of Children (NOC): NOC measures the number of direct subclasses to a class. Higher NOC in a class means that it may have more influence on the system design and therefore require more testing of the class and its methods, but it also comes with greater reuse of its methods. A high NOC also warns of possible misuse of sub-classing [11,12].

B. ISQ improvement studies in practice

The quality improvement studies chapter consists of short studies on techniques and methods to improve the ISQ of a system. All these studies are connected to metrics or technical debt in some way.

Static analyzing tools do in some cases collect metrics, in other cases compares code against metrics or both. Coding standards and quality gates often use metrics; coding standards use them as guidelines on how to write the code (e.g. should be less that 400 LOC for each method) and quality gates use metrics to compare the code so that the code clears the minimum requirements (e.g. a method got a CC of less than 10).

Technical debt is widely used nowadays, it is very useful to connect quality to measurable concept thus it can benefit in measuring and understanding software quality. Unit Test and Code Coverage and organizational structure metrics are metrics in them self but are not collected within the study, instead they are worth to mention as important metrics.

Knowledge sharing is an important topic when discussing ISQ because some errors and bugs exist because lack of knowledge when programming. Knowledge sharing can be done in different ways; one way is through review of code, metrics can help choose what code to review by showing if there exists code that have to many lines of code or too high complexity etc.

At the end of the study, those improvements studies are presented to GTT as further actions to improve ISQ.

1) Coding standards

Another name for coding standards, is coding conventions. Coding conventions are styles and coding guidelines. It is a way to write code so that code is consistent in a project. Some aspects the coding convention brings up/guide programmers to write are [14,15]:

- Naming
- Comments
- Formatting
- Classes
- Indentations

At Google they express the importance of the coding conventions in C++ because of many features in the coding language. They continue by mentioning that this can make the coding language complex which increases the risks of bugs and makes it more difficult to maintain [14].

Further reasons to implement and heed the code conventions are mentioned by many; Java, ISO 9000 and the Capability Maturity Model (CMM) amongst them. Below are reasons why conventions are important and should be used [15,16]:

- There will be less common type errors
- Many different programmers will maintain the program
- It will be ported more easily to other operating systems
- It is easier to read and understand
- The style will be more consistent
- The cost of software is 80% due to the maintenance

ISO 9000 and CMM go as far as saying that coding standards are mandatory for any company that has any sort of quality goals [16]. It is also said that perhaps the first and easiest way to improve ISQ is to introduce a reasonable coding standard [17].

2) Quality gates

Quality gates are controls on code which works as an extra step after implementation of a feature before it can be marked as complete. For a feature to be declared as complete when using quality gates the feature has to pass all the criteria that are established beforehand by the team [18,19]. Here are five examples of criteria that Microsoft has established in one of their projects [18]:

- No unit test created for the code may fail
- 80% code coverage has to be met for the unit tests
- The public methods shall have documentation

- Code that does not have unit tests should have no errors or warnings from static analysis tools
- The build must compile on the highest level and it shall not get warnings or errors

These criteria are specific for one team in one project at Microsoft; there can be more or less criteria depending on the team decisions. Quality gates is a technique that has been created with the goal of increasing software quality and is one that has been applied successfully for assuring quality in a piece of software code [18,19,20].

The use of quality gates increases implementation time but it also increases the quality of the code. It can also improve communication within the team depending on how the controls of quality gates are done [18,19]. Furthermore, features do not get implemented and labeled complete before they are robust enough to handle real-life activities [18].

As quality gates can be seen as best practice and therefore also include that the coding conventions should be followed, errors and faults in the code should be found earlier. Making code following coding conventions and having unit tests etc. makes the code more maintainable and reduces the cost of the whole project [21].

3) Static analysis tools

Static analysis is a tool used when controlling code in different ways and when to find defects early [22,23]. There exist different types of analyses that can be done; syntactic analysis, data-flow analysis and flow-graph analysis. Syntactic analysis checks code against earlier defined patterns like coding conventions. Data-flow analysis monitors variables and their states in all flows which predict null-pointer exceptions, not closed database objects in flows and more. Flow-graph analysis checks complexities of methods; this complexity is called CC (see *A. Metric Studies*) [22]. So by using static analysis tools metrics on the system can be gathered, code can be matched against patterns and reviewed for defects and bad habits of programmers [22,23].

There are many benefits from using static analysis tools [22,24]:

- Early bug detection
- Improving development productivity
- End-product reliability
- Enforces coding conventions
- Making maintainability easier
- Increases quality
- Finding potential performance issues
- Finds design defects
- Reduces need of manual coding reviews for finding cosmetic defects
- Makes review process more manageable and predictable
- Helps increasing the security of the software

4) Technical Debt

Technical Debt (TD) is a metaphor which stands for potential risks and cost of neglecting standards. The standard should determine the minimum quality requirements of the system (e.g. a method should not have more than 400 lines of code). The standard could cover many different aspects of software quality, e.g., coding conventions, test coverage, comments, architecture, review coverage etc.

The concept of TD is first mentioned in Cunningham's paper: "Neglecting the design is like borrowing money. Refactoring, it's like paying off the principal debt." [2] Seaman and Guo have given out their own explanation of TD: "Technical debt is a metaphor for immature, incomplete, or inadequate artifacts in the software development lifecycle that cause higher cost and lower quality in the long run." [25]

During this study, TD has been defined as a measure of how much a piece of code is deviating from standards which the company has set up for their source code. In this study, the calculation of TD has been simplified into the effort of fixing a certain amount of code with "poor quality" to code with "good quality". Here, "poor quality" and "good quality" were referring to code-standards set by the company.

"Developing slower because of this debt is like paying interest on the loan. Every minute spent on not-quite-right-code counts as interest on that debt." [2]. As long as TD exists in the project, there will be more cost and resources needed to maintain the project. The interest of TD stands for the additional cost of maintenance is added to the Accumulated Technical Debt (ATD) as well. Thus we can introduce a TD Interest Rate (TDIR) to the study. TDIR is another key issue to determine the ISQ. Their relationship can be explained as follow formula:

Formula 2: $ATD = TD + interest = TD \times (1 + TDIR)$

Usually TD is calculated in the unit of man-hours, man-days or man-months and from there translated over to a cost of how much this will cost.

An example of this can be that a company has a debt of 26 man-weeks and one man-week costs around 5'000 dollars, then the debt will sum up to: 26 x 5'000 which is 130'000 dollars. This TD is also calculated in percentage, how much the code deviates from the standards totally in percentage. This percentage can also be seen as an interest, because this percentage can be added to the time it will take to solve the deviations. When having an interest of 10 percent all fixes will take 10 percent longer time.

TD allows:

- To provide a common language for all teams.
- To monitor trends.
- To report comparative data.
- To proactively manage application asserts.

During this study, a customized calculation of TD was created. This customization used a paper from Seaman and Guo [25] as a theoretical background.

The TD calculated in this study consists of two parts: the cost of fixing a deviation from standards and the interest caused by the debt during maintenance activities. The details of the calculation of this TD can be found below.

As mentioned above, to measure TD the team needs to have established a standard to follow. When the standard are decided a database is created with numbers for the time it takes to fix deviations, the meaning to do this is to know how much time the different deviations will take to fix, excluding the interest. This will be used when summarizing the time later on in the process of calculating the TD. The time should be decided in man-hours so that the estimation in the end will be as accurate as possible.

When time is decided upon, an average cost of a man-hour will be needed. This will be done if the team wants to put a

cost, a number of dollars on the debt, which is preferred for understanding of the importance. Otherwise the debt can be calculated in man-hours to man-years.

When these preparations are finished, the deviations from the standards are calculated. To acquire the actual number the time to fix all deviations is summed and multiplied with the man-hour cost.

The interest of TD can be calculated in many different ways. During the study, a customized method of calculating interest was adopted [25]. The interest was calculated based on the levels of deviations collected by metrics. The higher deviation level has been set with higher interest rate. Then the interest of each object (model, class or function) was calculated by multiplying the set interest rate and the deviation. The total interest was the sum of all interest from all objects. TDIR can be calculated according to Formula 2, it can be predicted that the higher TDIR implies higher cost in maintenance in the future.

TD can be easily understood by managers without going into too many technical details, since is directly shown in a form of cost, important enough to catch attention. Comparing TD values of different period of a project is a good way to monitor the ISQ trend of the life-cycle of that project [25].

Using TD is also a good way for developers and employees in the lower part of the hierarchy in the company to show the higher-ups, the managers, that a quality check is needed or that the project needs more time to be finished. TD can be used by developer to verify their work as well [25].

5) Unit Tests and Code Coverage

A unit test (UT) is a test written by the developers or testers to white-box test the code in the earlier stages of developing [26].

UTs are used for finding bugs early, before the code goes to the testing phase [26]. Having established UT they can be relied upon when changing the code, the reason of this is because if the UT does not fail then the code is still working and can be sent to the test department or testers, but if it fails then the code needs to be rewritten. To be able to rewrite code which has faults before sending it to the testers can save lot of time and money [8].

Code coverage indicates the percentage of code that has been covered by different tests (sometimes it only refers to UT). Code coverage is a term used for many different forms of coverage in code; it can be statement coverage, branch coverage or an indicator telling how many lines of the code which was covered [26]. The ideal code coverage percentage should lie between 70 to 90% where more trivial modules can be at 70% and critical modules at 90% [8].

6) Organizational Structure metrics

Nagappan et al. [27] made a study about the influence organizational structure has on software quality and created some metrics to present it. Organizational structure metrics are metrics focused on the organization and the developers of the software. The metrics contains information on how many developers had edited the measured code, if there is a master of the code (a person that has made more than 75% of the editing of the code), etc. The results of these researches indicate that the measurement had around 86.2% failure proneness [27].

7) Knowledge management

Knowledge sharing regards sharing information on techniques for coding, information of standards, knowledge on the written code. Knowledge sharing can be done using different methods, for example; code review, peer review and pair programming.

Code reviewing takes place when a group of developers take a part of code, it can be a method or a class or just some rows, it depends on the developers or if they use some special method for reviewing. The code is looked at manually by the developers and discussed between them. Usually code reviews are done to code known to have poor design, and/or do not follow best practices, or to find code with such [28].

Peer reviews are done through letting a person, with similar expertise as the creator of a snippet of code, doing an assessment on the code. Peer reviewing is a short version of code reviewing and can find many bugs or errors undetected by the developer who wrote the code [29].

Pair programming is done when two developers work together on the same code. The method of Pair programming can be modified to fit the organization way of working, but usually there are two developers who take turns between the roles “driver” and “navigator”. The driver is the one who is actively writing code on the computer and the navigator is the one who watches the driver work, constantly identifying defects and coming with suggestions [30].

C. Early studies on MDD quality and metric collecting

Heijstek and Chaudron [57] have done a quality study in a large-scale MDD process, which gave a good overview of relations between metrics and software quality. However, only model size and complexity related metrics was discussed in their study. Their study only analyzed connections, no concrete solution was provided.

Staron and Nilsson [34] presented a framework to build measure system for software quality [34], which has inspired the study in many aspects: i) automation of metrics collection; ii) configuration options for stakeholders; iii) integration of current infrastructures.

Monperrus et al. [35] has conducted a case study about model-driven engineering metrics for real-time systems [35]. The study from Monperrus el. had different focus than this study, metamodels were used to avoid big cost and redundancy; however, it still gave some inspiration on measurement of models. Furthermore, it also pointed out that to minimize the cost, measurement tool should focus on one particular area.

Another source of inspiration is a position paper by McQuillan and Power [36]. The paper didn’t provide any industrial evaluation, and their observations are very practical. For example: “Observation 4. We can ‘lift’ code metrics to the model level”. The suggested metrics also contributed to the study.

Lange [37] discussed the importance of “Model Size Metrics” and what kind of metrics to be used in MDD [37]. This study showed the importance of model size metrics and is the reasons they have a role in this study.

At last, the EMISQ method discussed by Plösch el. [38] is not related with MDD, but did offer a good model of measuring ISQ.

None of the studies mentioned above used TD as an intuitionistic expression to relate software quality to concrete value in reality.

III. Methodological Design

As shown in Figure 1, the study was divided into three parts: investigation, implementation and evaluation. Investigation means collecting information, analyzing a problem field and determining requirements. Those requirements with higher priority which were considered feasible under particular circumstances were implemented during the implementation phase. Finally, the evaluation part offered support for the metric tool through both customer feedback and statistical analysis.

Initially an execution phase was planned but was merged with the evaluation phase as the scope and schedule of the thesis project evolved. An additional reason for eliminating a separate execution phase was the long duration (2-4 months) of the development life-cycle at the development team. Time constraints of the project made running both execution and evaluation process unfeasible.

A. Investigation

This phase had the purpose to set the scope and requirements of the study. ISQ is a wide and open topic and many factors could affect ISQ [39]. To narrow down the scope and set a feasible goal, much research work needed to be done early on, including literature reviews, discussions, meetings and surveys among managers, testers and developers in the development team.

Literature reviews and discussions offered theoretical basis and a choice of ready approaches to consider. Meetings and surveys revealed key problem areas, at least thought of as such by team members. By combining results of different research methods, a list of approaches to improve ISQ could be identified and implemented. Basing on that information, a collection of suggestions was presented to competent personnel at the development team.

Software code metrics (the measurement of the software product and the process by which it is developed [1]) was decided as the main focus of the study during the investigation part.

B. Implementation

The MetricAnalyzer is a Java™ application which uses IBM ® Rational Rhapsody (hereafter shortened to just Rhapsody ®) Application Programming Interface (API) to collect various metrics results from software models and then generates a report on ISQ of the analyzed software. It can also be used to compare historical statistics to generate a report on ISQ trends for a specified time period.

The implementation of the metric tool MetricAnalyzer was vital for the study. First, without implementation of a running program/tool, the result in the study would lack substance. Second, the improvement of ISQ is a continuous and step-by-step process, which needs support from automated measurement and report tools. Finally, the costs of running measurements without an appropriate code program/tool are considerable. To initiate the implementation phase, a workshop was held with the participation of several key members of the development team. Participants discussed various suggestion points from the investigation part. Some feasible solutions were also discussed. The outcome of the workshop included a prioritized list of metrics to implement, a prioritized list of functionalities and other requirements (see Appendix C for detailed outcome for the workshop).

Many "Agile Software Development Processes" characteristics such as small deliveries, frequent customer meetings and iterative improvement based on feedback were introduced during the implementation phase. This is partly because many requirements were not fixed beforehand. Limited time and resources was another reason for elaborating an intensive development process that could quickly adapt to changes.

C. Evaluation

The evaluation phase had two parts: feedback collection and statistical analysis. Interviews with related personnel and an anonymous survey were the main element of the feedback. Historical statistical results are collected from the tool and compared with project bug numbers. This offered a numerical and objective basis for the evaluation.

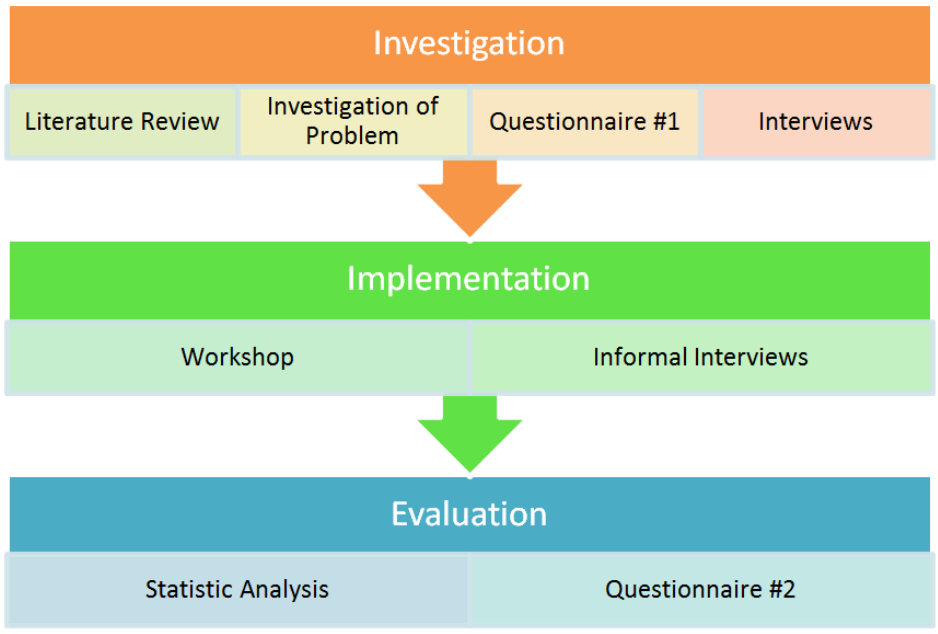

Figure 1: Methodological design overview

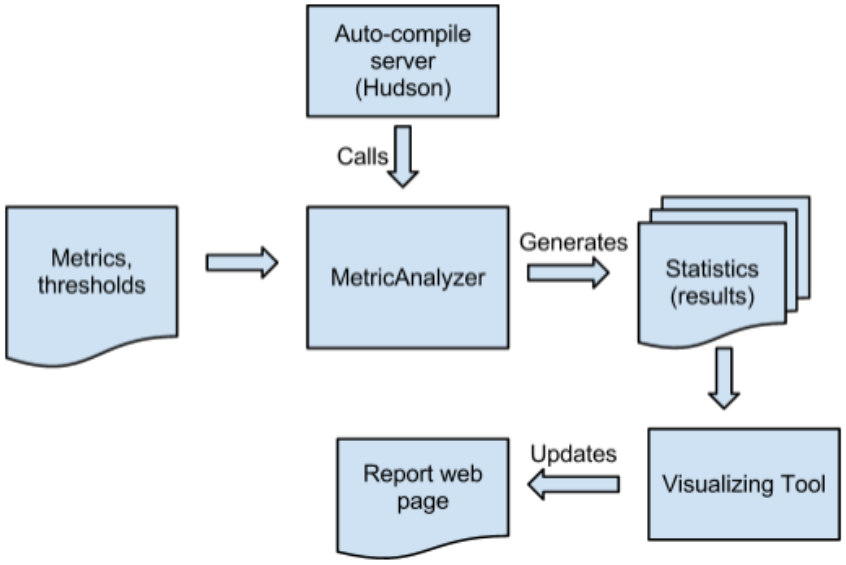

Figure 2: The Framework used in this study to analyze ISQ

Statistical analyses proved that the result from the framework can reveal the ISQ to some extent. Feedback collection also gave a positive estimation on the effort of the framework. Details of evaluation could be found in chapter V Result and Analysis.

IV. Solution and Implementation

The framework built in the study consists of many parts: a collection of metrics, thresholds for metrics, the MetricAnalyzer tool to calculate TD, visualizing tools to show the report and the integration with auto-compile server. A brief structure can be seen in Figure 2.

A. Developing environment

The development team is developing an embedded system for trucks. This system is developed using the model-driven environment (MDE) Rhapsody ® (for more information about Rhapsody ® see IV.A.2 Rational Rhapsody ®), which means that the development team is using a model driven development technique. There were no findings of metric gathering tools that could be directly integrated into Rhapsody ® within the allotted time. Therefore, based on that and on the findings of the organized workshop it was decided to implement the MetricAnalyzer tool using the API that comes with Rhapsody ®. During the workshop and interviews it was decided that using only plain text was not good enough as a report and that the report would rather be the form of a webpage showing the most relevant information. This webpage was constructed with the JavaScript ® library Data-Driven Documents (see IV.A.4 Data-Driven Documents (D3) for more information).

1) Model-driven development (MDD)

"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing." [41].

Working with MDD is one of the bigger steps made in software programming since the compiler was released. It is used to create a more abstract level of systems and to help understanding more complex software system [42,43]. A general aim of MDD is to create an abstract model of a system. In some of the models code for specific purposes can be implemented. Lastly in MDD a tool is used to auto generate source code from the models [44].

2) Rational Rhapsody®

Rhapsody ® is a big tool of great functionality and support for software engineers and software developers in their work. The biggest highlight of Rhapsody® is that it provides system engineers and software developers with a MDD environment for real-time and embedded software which is based on UML. Furthermore, Rhapsody® generates applications/systems in various programming languages including: C, C++, Java and Ada. When generating the behavioral and architectural view is also included [45].

3) Rhapsody® API

Rhapsody® API accompanies the installation of Rhapsody®. This API exists for Java™ programming and for COM and allows creation of applications that can access and modify Rhapsody® model elements. Accessing and modifying

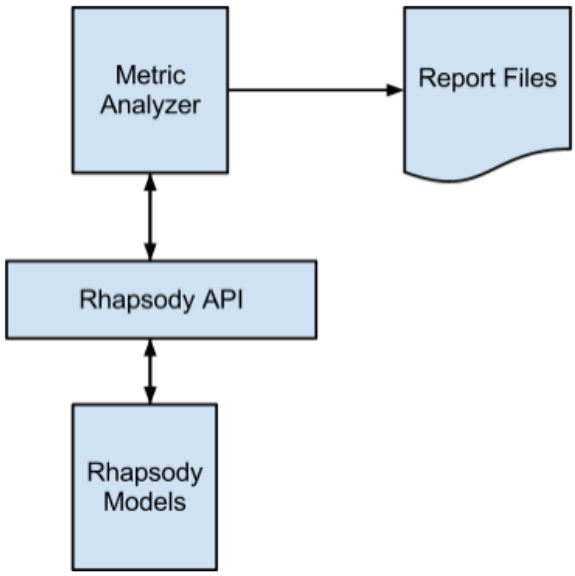

Figure 3: Data Model of MetricAnalyzer

implies reading, changing, adding and deleting of all model elements that exist within the Rhapsody® Browser [46].

4) Data-Driven Documents (D3)

"D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG and CSS." [47].

D3 allows and helps to bind data to a Document Object Model (DOM) and to visualize data, either in the form of bar-charts, pie-charts, creating/populating tables etc. It also allows interaction in the form of mouse action or keyboard actions and smooth transitions to the visualizations [47].

B. Implemented tools

This chapter focuses on the implementation of two parts; a Java™ program called MetricAnalyzer which analyzes models and a web page which summarizes and shows the report created from the result of the MetricAnalyzer. The description and reasoning for implementation and design can be found in this chapter.

1) MetricAnalyzer

MetricAnalyzer is a program written in Java™ which analyzes models in Rhapsody® with the help of Rhapsody® API. It reads configuration from a customized configuration-file and then generates results for different metrics. The results are put into CSV files and TXT files for the report program to read.

a) Overall design

MetricAnalyzer works closely with Rhapsody® API. Rhapsody® API enables the program to read data and statistics from Rhapsody® models needed by different metrics. With the data and statistics gathered from Rhapsody® models, the MetricAnalyzer filter out useful information and generate readable data according to needs from metrics and configuration. The readable results are stored in generated CSV- and TXT-files.

b) Detailed design

A detailed data flow design can be found in Figure 4. A description of all sub-models can be found in Table 1.

Table 1: Sub-models and descriptions in MetricAnalyzer

Sub-model name	Description
UI (User Interface)	This is where the program interacts with users. A simple command line handler which takes a command parses it and sends parameters to Engine if the command is complete. It also prompts help messages, error messages and other event messages.
Engine	Works closely with UI and runs Model Analyzer.
Model Analyzer	Reads data from Rhapsody Connector, interacts with metrics, filters out useful information and then generates result data.
Rhapsody Connector	Helps Model Analyzer to connect with Rhapsody® models through Rhapsody® API.
Metrics	This is where all the metrics are calculated, every metric class represents a specific kind of metric, they parse information passed from Model Analyzer and send back results after their own functions been executed.
Configuration	Handles user-configuration and metric-settings, gives the settings to Model Analyzer and metrics when needed.
Utilities	Gives general support for all other models, offers functionalities such as file-accessing and logging.

An end-to-end data flow for the MetricAnalyzer can be described as follow: UI gets user command, parses it, and then sends it to engine if the command is correct and complete. Engine calls the Model Analyzer as the command indicates and

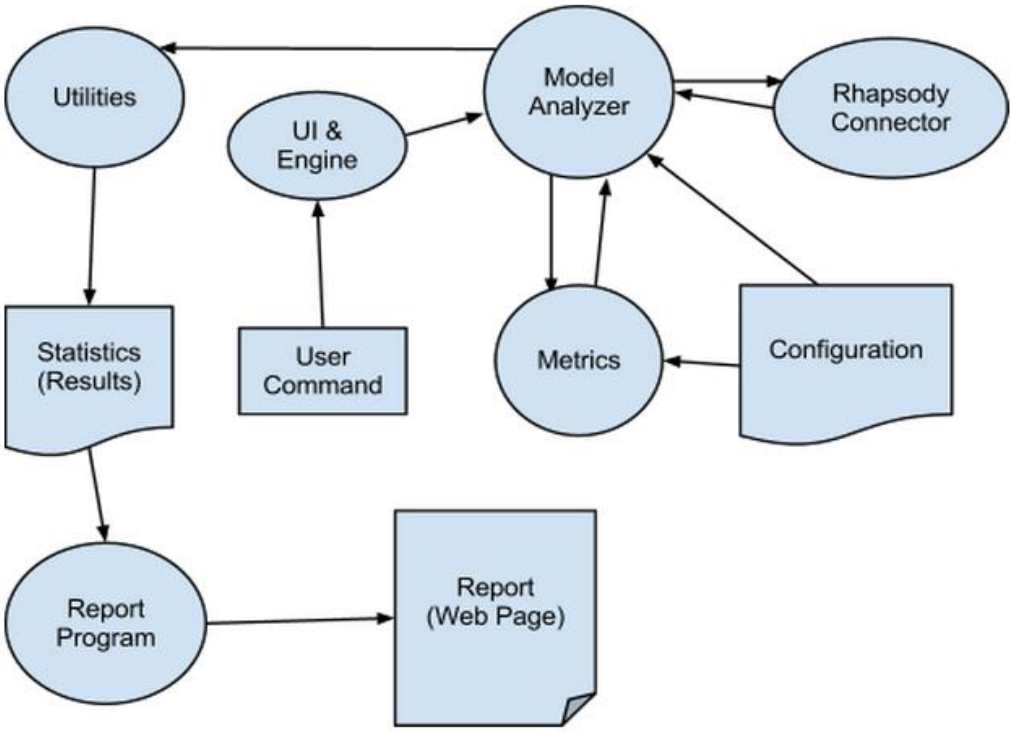

Figure 4: Data flow of sub-models of MetricAnalyzer.

passes the parameters. The Model Analyzer reads configurations and initiates metrics according to information in configuration file. After the metrics are ready, the Model Analyzer connects to Rhapsody Connector, goes through every project found in Rhapsody® and reads packages and classes from models. While Model Analyzer is processing data from Rhapsody Connector, it also checks settings from metrics. For example, Model Analyzer is dealing with package A, it checks with every metric, it only sends details of package A to those metrics which need package data and skips others. After the Model Analyzer has processed all the projects, it summarizes all the data collected, filters useful data and writes it into files through Utilities. Utilities also handle other operations such as reading/writing files and managing data, which can be very big. At last, the results are written into CSV- and TXT-files, the report program reads the results and then generates a graphical web page which contains different charts and forms to visualize the results.

c) Metrics Applied in MetricAnalyzer

Metrics were taken from two papers [11,12], these were both older metrics and also metrics constructed for OOP, for more information on the metrics that were chosen, see II.A Metric studies. Firstly, the metrics were mentioned in the interviews for personal opinions from the key-persons. Secondly, they were brought up in the workshop to create a discussion and discern a prioritized list indicating which metrics that would be implemented in the MetricAnalyzer, which metrics that had the highest prioritization of implementation and also which thresholds they would have.

Before the metrics could be included in the MetricAnalyzer many of them had to be discussed and reconstructed to fit Rhapsody®. All metrics were gathered from within Rhapsody® with the aid of the Rhapsody® API.

After the Workshop two classical metrics were chosen; CC and LOC. Two metrics were taken from research papers about metrics for OOP; DIT and NOC. Lastly five were implemented based on ideas from the workshop and feedback, metrics that key-employees wanted. Of these five three can be labeled size metrics; size of statechart, number of non-constant attributes and numbers of methods, the last two were comment percentage of description and number of descendants. In total there were nine metrics which were chosen for the MetricAnalyzer, below are all described, if and how they were modified to fit MDD.

Cyclomatic Complexity (CC): CC is from the beginning a source code metric that was, in this study, configured to better fit Rhapsody®. In Rhapsody® there exist implementation tabs in functions and operations which can contain code. The functions and operations were looped through and CC was calculated for each implementation that had code. The CC is calculated through adding all “if”, “else”, “while”, “for”, “switch”, “||”, “&&” that could be found, into a variable.

Lines of Code (LOC): The modification that was made to CC was also made to LOC. The only LOC counted were the lines in the implementation tab. Furthermore all comments were excluded when counting the number of LOC.

Size of Statechart (SOS): The size of statechart is the sum of the number of all transitions, states and events in each statechart.

Comment Percentage of Descriptions (CPD): Usually the Comment Percentage (CP) metric is the percentage of code which is commented. For the team at Volvo GTT this was not the most important sort of commenting. The comments CPD was created to count were how many of the different model element that were commented in the form of having a description explaining what that element were doing. CPD has a minimum requirement of five characters to be counted as described.

Depth of Inheritance Tree (DIT): This metric is unmodified in its behavior, it counts how many steps it is from one class to the root class of the particular hierarchy tree.

Number of Children (NOC): The NOC is the number of classes that directly inherit from one class.

Number of Descendants (NOD): This metric was added to the list of metrics because the team at Volvo thought that it was needed, not because of findings in the literature. The NOD calculates all classes that inherit from one class or inherit from classes that inherit from that class etc.

Number of Non Constant Attributes (NCA): NCA calculates the number of non-constant attributes for each class. This metric was requested in the workshop.

Number of Methods (NOM): NOM was requested from the development team and as the name suggests it counts the number of methods for each class.

d) Configuration-file and thresholds

This sub-chapter contains the description and reasoning of configurations and thresholds.

In this study all deviations from standards got the umbrella-name violation. The range of violations has been divided into three levels (green, yellow and red) according to the values which been calculated for each metric. Red gives the highest TD, yellow gives lower TD and green gives zero TD. As an example: if the value for one metric is above the red threshold, the TD for that metric is red and the time to fix that violation is the number in the column “Time to Fix Red” (see Table 2). So if the result of CC for one function is 280 then the time to fix that violation is five hours, and to make the concept of TD easier, each man-hour counts as one TD.

Metric Name	Green Threshold	Yellow Threshold	Red Threshold	Top List Threshold	Time to Fix Yellow	Time to Fix Red
LOC	1	80	200	100	2	5
CC	1	7	10	10	2	4
NCA	0	10	20	10	1.5	3
NOM	0	10	20	10	4	8
CPD*	100	99	50	50	4	8
SOS	1	20	50	10	4	8
DIT	1	4	6	10	8	16
NOC	1	4	6	10	2	5
NOD	1	4	12	10	2	5

Table 2: Thresholds and TD for Metrics

**CPD is not included when calculating TD.*

The thresholds and "time to fix" are made by referring the Sonar standard and then adjusted to suit the MDD environment at the development team. The configuration-file was created externally so that the employees easily could go into the file and modify it without changing inside the code or the MetricAnalyzer. The configuration-file holds information on the thresholds and debt given from violations. Furthermore the configuration was created to be able to decide which projects within the system that would be measured. The development team wished for the possibility for packages, modules, and classes' etc. to be excluded from the measurement. This was wished because they were finalized and would not be modified again and therefore the data from those classes would be useless [10].

2) Visualizing Tool - Report Web Page

The tool was created based on the information collected and calculated from the MetricAnalyzer, and it had to be visualized in such way that the result could be easily interpreted. This was done by using the MetricAnalyzer reports to generate a visualized report, to construct an overall view and evaluation of the quality of the project, for more information on the evaluation see Chapter V Results and analysis. The program was created with the JavaScript library D3.js and this subchapter describes the visualizing program with a summary of its functions and parts. To read more detailed information on the information shown in the Visualizing Tool see Appendix A.

a) Main Page

The main page (see Figure 5) of the visualizing tool is the general information board which was constructed by many parts. Each part display results of different metrics which were collected from the models of the system which the development team worked on. It was designed to make users to be able to catch the overview of the project in a glance and to catch everything through one page. All values shown in the main page was calculated into TD.

b) Project details page

Project details page (see Figure 6) was created to give more detailed information for each project, specific information. All information shown in the details page was in raw values and not calculated to TD.

V. Results and Analysis

This chapter contains the results of the thesis, this includes; information gathering, results from comparing historical data and feedback from the development team on the framework with reports.

A. Information gathering from developers

To narrow the scope and help make the study fit the development team's goals and needs information gathering were carried out among developers. This was done by one questionnaire in the beginning and a workshop in the mid-time

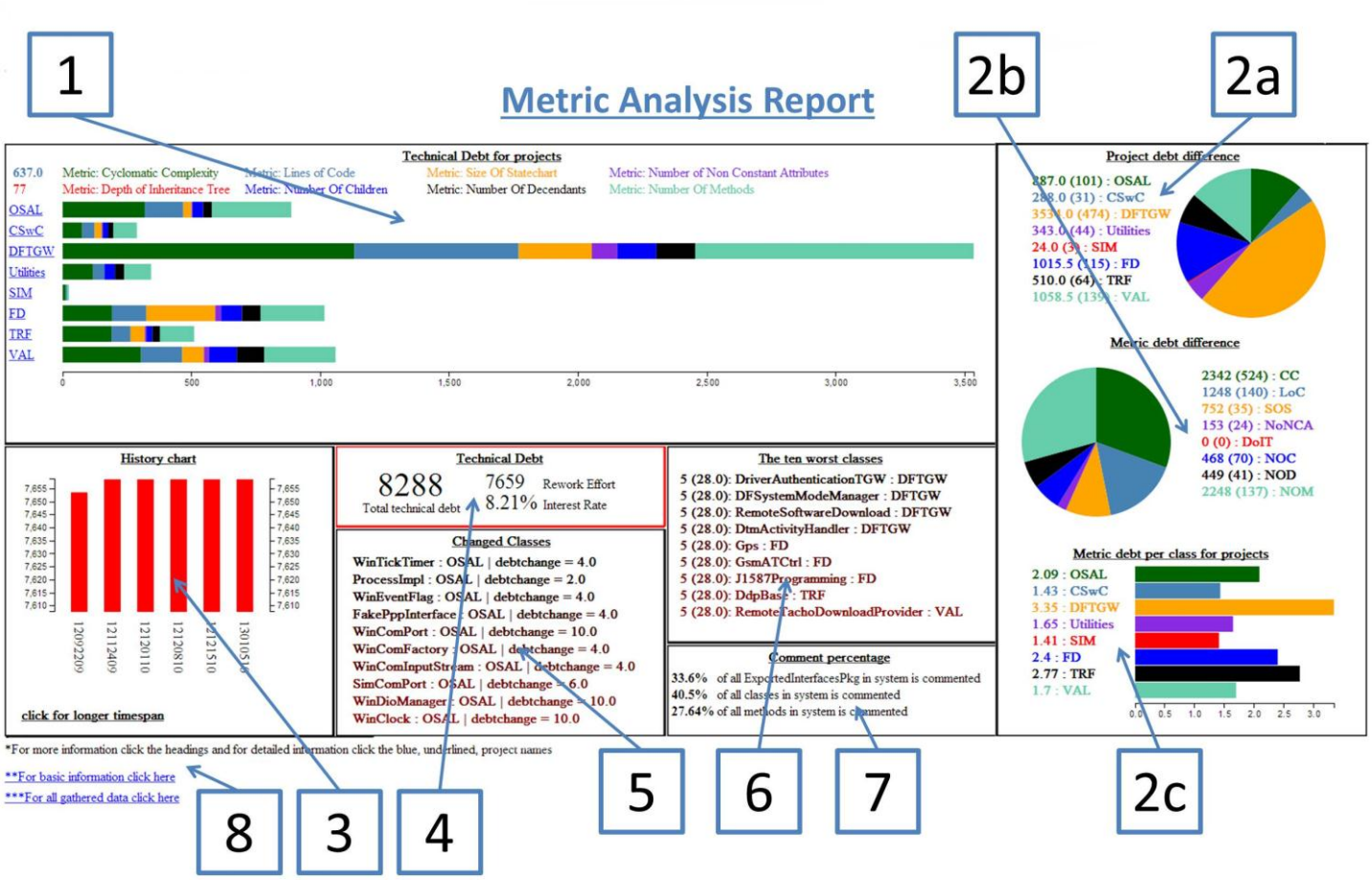

Figure 5: General Information Board of visualization program that is based on MetricAnalyzer report (1) TD for projects, (2a) System debt difference, (2b) Metric debt difference, (2c) Metric debt per class for projects, (3) History chart, (4) Technical Debt, (5) Changed Classes, (6) The ten worst classes, (7) Comment percentage, (8) Information.

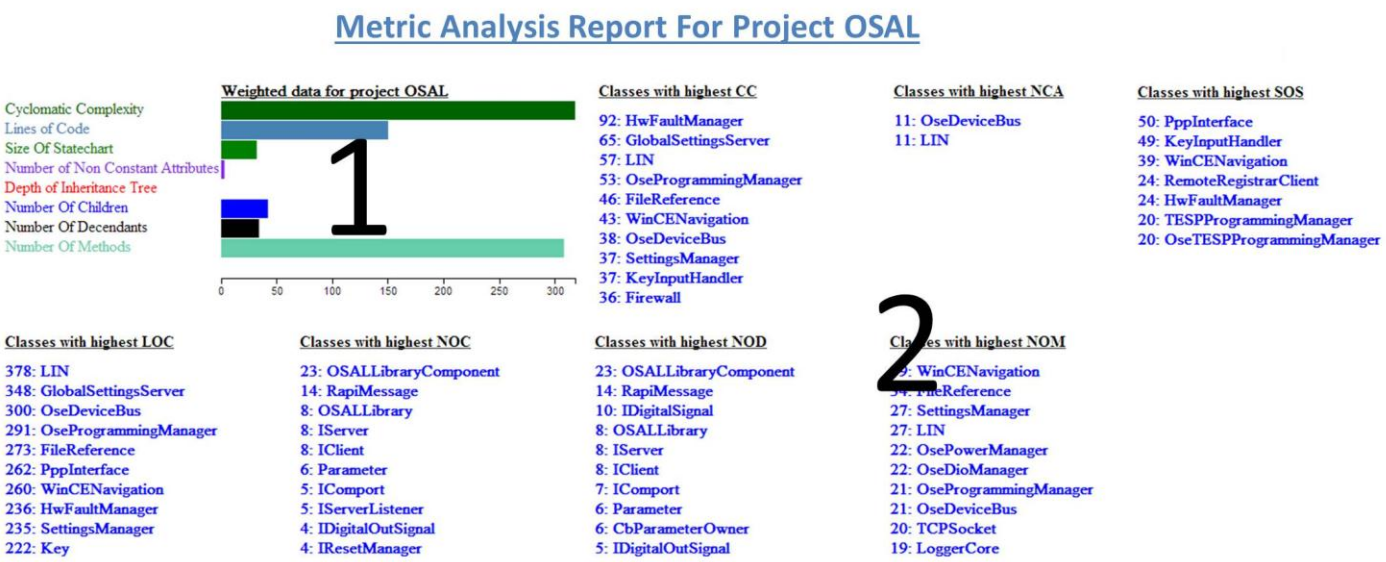

Figure 6: Detailed TD and metric information for projects
1. Bar-chart with weighted metrics, 2. Top ten classes within all metrics in that project

of the study.

1) Questionnaire

In order to analyze the existing problem and find out possible solutions, a questionnaire was created and sent out to all employees in the development team (testers, managers and developers). The questionnaire was also made to get some information on what the staff wanted to do to improve ISQ. In the questionnaire there were some choices they could choose but also text areas where they could write if no choice suited them. The questionnaire was sent out to 36 team members of which 23 answered.

For the multi-choice questions "What is software quality to you?" and "How would you measure software quality if you can decide?" the most popular answer was "fulfills the functionality requirements" and "bug number" in that order, for the most frequent answers see Table 3 and Table 4.

Table 3: Question answers for questionnaire

Question: "What is software quality to you?"	
Answer choices	**Nr votes**
"fulfills the functionality requirements"	16
"Code is easy to understand and has clear logic"	13
"is easy to maintain"	13
"is bug free"	13
"good performance"	9
"everything is well documented"	5

Based especially in Table 3 it can be seen that the employees in the development team at Volvo GTT wanted tidy code that was easy to read and maintain.

Table 4: Question answers for questionnaire

Question: "How would you measure software quality if you can decide?"	
Answer choices	**Nr votes**
"bug number"	20
"unit test coverage, test coverage"	19
"complexity of code logic, connections between classes"	15

For the question "What do you think about current product quality? Please give a score from 1 to 10". The average score was 5.3 out of 23 answers. From this it can be seen that there is big potential for the quality of the product to increase.

A summary was made from the questionnaire and with the summary as a start, information was gathered which included methods and implementations which were feasible to implement in the period of the study. Because of limitation some of the topics had to remain as suggestions to the team at Volvo GTT. For example, one suggestion was to increase UT coverage and test coverage. What can be seen in the questionnaire that also got captured in the framework was the interest of clean code, code that was maintainable and had few code smells.

Furthermore, through semi-formal interviews, needs of tools to show the product have been revealed. More information on the questions and answers in the questionnaire can be found in Appendix B.

2) Interviews

The questionnaire helped to capture a draft of the problem. To make the draft more focused on a smaller subject area, interviews with key team members were carried out.

The most mentioned areas/topics in the interviews were:

- Every participant thinks software quality, especially ISQ, is important for the product.
- More code reviews are needed.
- To implement metrics measurements for system.
    - Using metrics as a means to know which code to look at in the code reviews.
- Working more with UT and increasing UT Coverage.
- More knowledge sharing in many in form of: code reviews, pair programming and educations.
- To be more careful with coding guidelines and best practices.

- That TD could be good but hard to implement.

3) Workshop

The workshop was, in contrast to the questionnaire, held to get some more specific information on what to do in the next step in that phase in the study. It was done by inviting the key-personnel in the development team to two workshops on two hours each. Within these four hours quality improvement studies were presented, some were discussed and some were only brought up as information for the attendees. The main topic in the workshops was to go through the backbone of the metrics and see if there was anything which was to be changed in them. Furthermore, if there needed to be additional metrics or less of them and how to create the metrics so that they could be used within a MDD environment. The result of the chosen metrics was introduced in chapter IV.b.c Metrics Applied in MetricAnalyzer. There were discussions on how to handle lack of findings when it came to freeware tools to calculate metrics for MDE. It was also discussed that a possibility would be to create one with the help of the Rhapsody API. The discussion continued with how the information should be shown, which resulted in a website. There the information would be displayed so developers could go there and watch the development of the system through the metrics. The summary of the workshop can be seen in Appendix C.

B. Comparison of Historical Data

The development life cycle in the studied environment has been divided into Work Periods (WPs). The investigation of relation between TD and bug numbers was done based on six selected WPs. In Figure 7, total TD was collected for each of the six WPs to visualize the TD over time for the system.

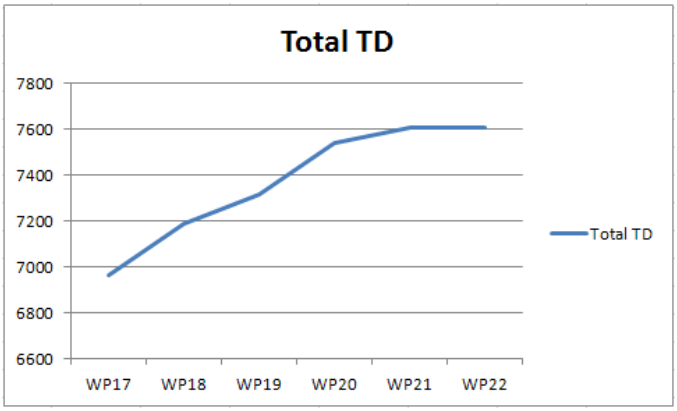

Figure 7: Total TD for Work Periods (WPs).

In order to find out relationship between TD calculated in the study and ISQ, an investigation was done in the active WPs. Increase of TD for each WP (total TD in one WP minus the total TD of previous WP) and the number of bugs found in that WP was selected as key values to be compared. Figure 8 show that the two values have the same trend. An assumption was made based on the result: there is a firm relation between those two values. Thus, under normal situation, if the assumptions are right, whenever the introduced amount of TD has increased in one WP, then the bug number should be higher than the previous WP as well.

However, many issues should be considered, for example, time span of a WP, number of changes added in that WP and complexity of changes.

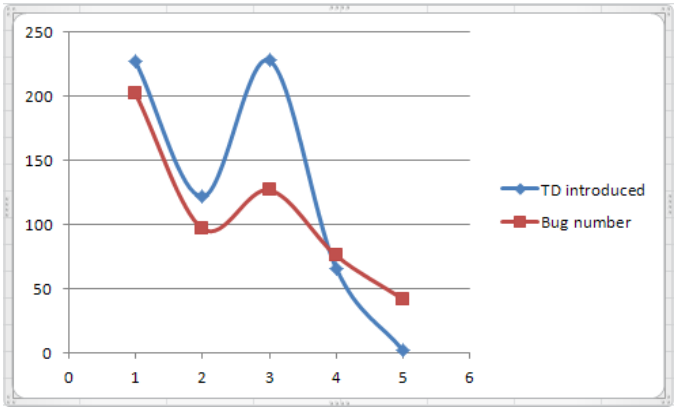

Figure 8: Relations between introduced TD and bug numbers in different WP.

Another important aspect of the framework was to show how the calculated TD, the ISQ, changed over time. Also, it was wanted to visualize how much it changed for each week, if it changed much and if it increased or decreased. Figure 9 is a history chart which shows the trend of the ISQ in the system. It can be seen in Figure 9 that compared to other measures it was introduced much TD in the system before the record "12082109". Such record is a possible indicator that there was modifications and added code containing high-risk code/designs checked in to the repository.

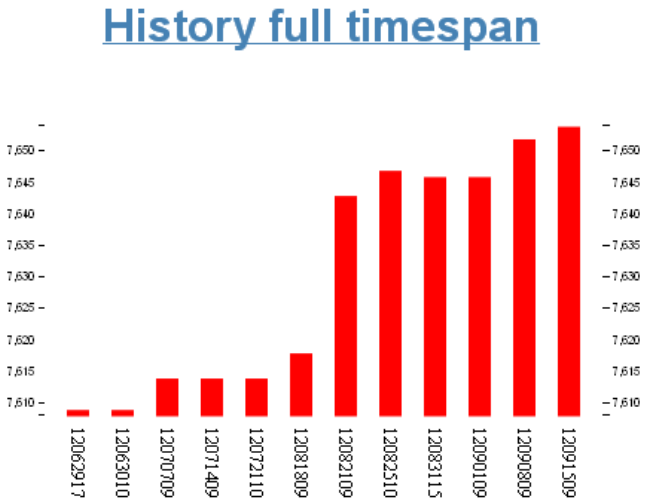

Figure 9: A chart in the report web page shows the increase of TD in a long term timespan.

C. Feedback of The Study from the development team

After the results were presented to the developers, a questionnaire was handed out to the audience who attended the presentation. There were 22 valid replies collected.

Among the 22 valid replies, 22 (100%) have answered "Yes" to the question of "Do you think the framework can help to improve ISQ?". To the question: "To what extent do you think the framework is helpful? Please choose a number from 5 - very helpful to 0 - not helpful.", 4 answered 5, 8 answered 4 and 9 answered 3. Only 1 answered 2 and no answers for 1 and 0. This makes an average result of 4.14. This shows that the team has confidence in the framework and positive expectation of the result.

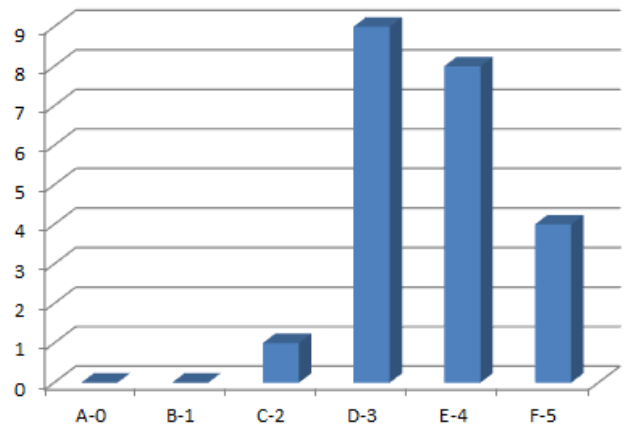

Figure 10: Replies to Question 2 in the questionnaire.

VI. Discussion and Further Work

This chapter brings up a discussion on the results, the reasoning behind different parts like the questionnaire and why certain metrics were chosen. Furthermore the discussion and future work discusses where this study can lead and further studies which can be done based on this study.

A. Findings and Importance of study

The primary findings of this study are: i) that the TD calculated from MDD models based on a collection of selected metrics has direct relationship with ISQ; ii) that the metric analyzing tool MetricAnalyzer combined with the report visualizing tool is efficient in tracking problem-prone models.

A secondary finding is showing the metric result in the form of TD with the support of a visualizing tool could help project managers to have a better vision and control over the project.

From the result, it also can be seen that the MetricAnalyzer's feasibility to predict bug numbers even before the testing process is positive.

ISQ is the first gate of software quality; it is the one of the keys to successful software projects. ISQ is vital in the long run of software development life-cycle and many researches have been done is this area. However, seldom research has combined software metrics (both basic and MDD oriented) and TD together in a MDD environment. This study carried out a valuable case study in that situation and the result is promising.

B. Alternative Explanation of Findings

Wilson [39] mentioned that some studies show that many existing metrics actually offer nothing more than the basic metric "Lines Of Code" [39]. The fact is that in the collection of metrics, which have been used by MetricAnalyzer, are related with size in one way or another, for example SOS, NCA etc. One possibility that can be seen based on Wilsons statement is that if the MetricAnalyzer only use the one metric "Size Of Model" (SOS + LOC), it would give out the same or even better result. However, more research is needed to prove those assumptions.

C. Metrics: reasoning and decisions.

The metrics which has been selected and applied in the metric analyzer tool MetricAnalyzer is the backbone of this study. The question of how to choose right/useful metrics from the many existing metrics emerged as a major problem. There are many experienced software engineers in the development team, the workshop was held with several experienced key-persons in the development team. In that workshop many questions was discussed: if the metrics were to be used, if they were useful, if there were needed some change to them, how the metrics could be useful in a MDE, and if there were metrics they wished for but was not in the list of metrics prepared before the workshop. This workshop helped filtering between the many metrics and to create metrics that suited the development team.

The list of metrics that were selected before the workshop and that was discussed during the workshop, originated from earlier defined object-oriented environment metrics. The main reasons for using these metrics during the workshop were that some of the metrics are easy to implement and understand, and the others are created and tested for measuring quality. These metrics were tested in earlier studies where many of them got positive results [11,12]. Concerns were raised that some of those basic metrics that were chosen would not be appropriate MDD. To cope with that, some metrics were customized and some metrics was created based on personnel's requests and added into the implementation.

D. Reasoning of Quality improvement studies

The quality improvement studies were done to give information on quality improvement other than metrics and code analyzing tools (see II.A.b Quality improvement studies). Due to time constraints and the fact that, there were many changes that could not be authorized, the quality improvement studies got low prioritization. Therefore these studies were made to notify developers and managers of methods that exist to improve quality.

E. Validity of study

The metrics that has been chosen as the backbone of this study are chosen from studies which has tested them in projects [12] or proposed by the Software Assurance Technology Center [11]. By assuming that these papers has some validity it can be assumed that the result in this study has a certain degree of validity.

Another important aspect with metric validity is that an important factor is what the stakeholders wish. This means that the metrics may be useless unless someone is surveying them and that they are used [10]. The metrics used are discussed, each one specifically, and only chosen if seen appropriate in the developing environment. Based on that information the metrics gets further validity.

F. Limitation of current work and suggestions for further research

This section discuss things which can be done to improve the constructed framework in the future and researches which can be performed using this study as a foundation.

1) Further configuration options

The configuration-file of the MetricAnalyzer was made to be an external file so it would be easy for the company employees to adjust. This was done because the configuration created in the study was set based on information and configurations from other sources than the developers. Therefore the configuration file should be calibrated by key-personnel so it is customized specifically for their system. This customization is important for many reasons, one of the

foremost reasons is that parts in the systems which should not be measured by the MetricAnalyzer can be excluded using the configuration-file.

2) More rigid standard and more accurate numbers for TD

Standards and thresholds made and used for the MetricAnalyzer are created partially by referring the sonar standard [49], partially by some preferences from team members in the development team. And also from online sources (this were done knowing the problem of validity but with modification in mind) then modified by the authors to fit MDD. The configuration needs to be changed by the developers and testers in the team. The reason for them to do the change is because their experience with coding and their system. For example; the time which need to fix different deviations, how high the thresholds should be, which parts of the system that is not supposed to be measured and other configurations that affect TD.

3) More customized metrics for rhapsody & MDD

The nine metrics applied in this study are partially decided within a workshop held with key-personnel and partly added during the implementation of the framework according to the requests from developers.

Due to uncertainty of the implementation-time of the framework, the decision made in the workshop was to implement the basic metrics first (see Chapter IV Metrics Applied in MetricAnalyzer). Then, if allowed by the available resources, more metrics would be added. Further reason for this was that the basic metrics are easy to understand by users who do not have the background knowledge of metrics. Thus, the process of adding and customizing metrics lasted till the late phase of implementing the framework.

Although the authors and the key-personnel in the team tried to include all metrics considered useful, there is a possibility that not all useful metrics were included. A potential research topic for the future is to search for more metrics suitable for their system and environment.

4) Test MetricAnalyzer on more projects

The results in this study are based on one project, using its timeline, to compare historical data with the number of increased bugs for each WP. More researches could be performed on projects using MDD to compare the result data against each other. Another valid test would be to test the framework against other projects also created in Rhapsody.

5) Further investigation on individual metrics

Future studies can be carried out on the metrics used in this study. This to see individually which metric has the most relevance when measuring ISQ.

6) More benchmarking of ISQ

Currently the result of the MetricAnalyzer only compares with bug numbers as can be seen in Figure 8. There is a common consideration that bug numbers is related with software quality but even so, researches based on other data could be carried out to get more results that can validate the results of the study.

VII. Conclusion

The topic of ISQ within MDD has been discussed before. However, this study has its own unique aspects. For example, it used TD to shown the result of model metrics and visualized the result into an auto-updating web page.

In this study, nine metrics has been applied in MDD. Some of them are basic metrics, some of them are OOP metric and some of them are specially made for the development team where the study was carried out. Based on the metrics, a framework has been built to measure and monitor the ISQ of the product constructed by that team. The results show that the TD calculated from those metrics has the same trend as the bug numbers over time on the same project. Thus, software metrics, which has been proved by at least two studies [11,12] in non-MDD environments, also can be used in MDD environment. The combination of the metrics used in this study can possibly give a promising result in other similar environments as well.

TD was also introduced to the team through the framework which was constructed during this study. The metric results needed a platform to be shown at and in a form that was readable and understandable for both developers and managers. TD is a metaphor which describes the result vividly and is easy to understand. According to the feedback gathered, the development team had a positive attitude for the combination of metrics and TD.

The visualization tool (the auto-updating webpage) that shows the result in a webpage with different chart is vital to the study, developers and project managers at that team. Team members and project managers have given positive feedback on using TD to track the ISQ and monitor the development process through the visualization tool. Without the visualization tool, the result would be harder to understand and therefore it plays a crucial role in monitoring ISQ. Furthermore the visualizing tool makes the information easier to access and used

This study got positive result from both analyzed statistics and feedback from related personnel, and in the future, further studies should be carried out to improve the framework with more metrics and validation tests.

The framework created in this study can be a good start for further studies with focus on ISQ of MDD.

References

[1] Mills, E. 1988. Software Metrics, Curriculum Module SEI-CM-12-1.1. Software Engineering Institute, Carnegie-Mellon University.

[2] Cunningham, W., 1992. The WyCash Portfolio Management System. OOPSLA '92.

[3] Briand, L., Wüst, J., Daly J., W., Porter, V., 2000 Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems, Journal of Systems and Software, 51, pp. 245-273.

[4] Geoff, D., 1996. Cornering the Chimera, Australian Software Quality Research Institue, EE Software 0740-7459/96.

[5] Kearney, J., K., et al., 1986. Software Complexity Measurement. Commun. ACM 29(11) pp. 1044-1050

[6] Robert N., C., 2005. Why software fails, IEEE Specfurm article.

[7] Jorgensen, M., Molokken, K. 2006. How large are software cost overruns? A review of the 1994 Chaos Report. Information and Software Technology 48, 4.

[8] McLean Hall, G., 2010. Pro WPF and Silverlight MVVM: Effective Application Development with Model-View-ViewModel. New York:Apress.

[9] Fenton, E., N., Neil, M., 1999. Software metrics: success, failures and new directions. Journal of Systems and Software, 47(2-3), pp.149-157.

[10] Westfall, L., 2005. “12 Steps to Useful Software Metrics”, The Westfall Team, Plano.

[11] Rosenberg, L., H., Hyatt, L., E., 1997. Software Quality Metrics for Object-Oriented Environments, The Journal of Defense Software Engineering, STSC.

[12] Kumar, S., A., Kumar, T., A., Swarnalatha, P., 2010. Significance of Software Metrics to Quantify Design and Code Quality. International Journal of Computer Applications [e-journal], 11(9) Available through: .ijcaonline.org [Accessed 17 July 2012].

[13] Henderson-Sellers, B., 1996. Object-oriented metrics: measures of complexity, New Jersey: Prentice-Hall.

[14] Weinberger, B. et al., n.d. Google C++ Style Guide.[online] Avaliable at: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml> [Accessed 29 August 2012].

[15] Sun Microsystems, 1995. Code Conventions for the Java™ Programming Language. [Online] Available at: <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html> [Accessed 29 August 2012].

[16] Paoli, S., 1999. C++ Coding Standard Specification. [Online] Available at: <http://pst.web.cern.ch/PST/HandBookWorkBook/Handbook/Programming/CodingStandard/c++standard.pdf> [Accessed 29 August 2012].

[17] Quantum Leaps, LCC, 2008. Application Note C/ C++ Coding Standard. [Online] Available at: <http://www.state-machine.com/doc/AN_QL_Coding_Standard.pdf> [Accessed 29 August 2012].

[18] Williams, L., Brown, G., Meltzer, A. and Nagappan, N., 2011. Scrum + Engineering Practices: Experiences of Three Microsoft Teams. Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium, pp.463-471.

[19] Ambartsoumian, V., et al., 2011. Implementing Quality Gates Throughout The Enterprise It Production Process. Journal of Information Technology Management Volume XXII, Number 1, pp. 2028-2038. Available at: <http://jitm.ubalt.edu/XXII-1/article3.pdf> [Accessed 29 August 2012].

[20] Pasi Ojala. 2010. Industrial experiences of developing quality gates for software development process. In Proceedings of the 4th WSEAS international conference on Computer engineering and applications (CEA'10), Stephen Lagakos, Leonid Perlovsky, Manoj Jha, Brindusa Covaci, Azami Zaharim, and Nikos Mastorakis (Eds.). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 33-37. Avaliable at: <http://www.wseas.us/e-library/conferences/2010/Harvard/CEA/CEA-03.pdf> [Accessed 29 August 2012]

[21] Laskowski, J., 2009. Increase quality and decrease costs with IBM Rational quality gates. [Online] Available at: <http://www.ibm.com/developerworks/rational/library/09/qualitygates/index.html> [Accessed 29 August 2012].

[22] Chaturvedi, A., 2005. Java & Static Analysis. Dr. Dobb’s Journal, 30(7). pp 25, 27-29. Available at: <http://search.proquest.com.proxy.lib.chalmers.se/docview/202692862> [Accessed 29 August 2012].

[23] Brew, W., Johnson, M., 2001. Value lattice static analysis. Dr. Dobb’s Journal, 26(3), pp. 30-38. Available at: <http://search.proquest.com.proxy.lib.chalmers.se/docview/202690314> [Accessed 29 August 2012].

[24] Chess, B., McGraw, G., 2004. Static Analysis for Security. IEEE Security and Privacy, 2(6), pp. 76-79. Available at: <http://ieeexplore.ieee.org.proxy.lib.chalmers.se/stamp/stamp.jsp?tp=&arnumber=1366126> [Accessed 29 August 2012].

[25] Seaman, C., Guo, Y., 2011. Measuring and monitoring technical debt. Advances in Computers, volume 82, pp. 25-46.

[26] Thornton, S., Wang, Y., H., 2003. Software Testing, SENG 623 Software Quality Management. University of Calgary. Available at: <http://www.slideshare.net/Softwarecentral/software-testing-3744255> [Accessed 31 August 2012].

[27] Nagappan, N., Murphy. B., Basili, V., R., 2008. The Influence of Organizational Structure on software Quality: An Empirical Case Study. In: Proceedings of the 30th International Conference on Software Engineering. Leipzig, Germany, 10 - 18 May 2008.

[28] Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Proceedings. Ninth Working Conference on Reverse Engineering, 2002, pp. 97-108, IEEE Computer Society.

[29] Software Peer Review Guidelines, 2007. Science Infusion Software Engineering Group (SISEPG). National Weather Service/OHD. [online] Available at: <http://www.nws.noaa.gov/oh/hrl/developers_docs/Software_Peer_Review_Guidelines.pdf> [Accessed 15 October 2012].

[30] Nagappan, N., Begel, A., 2008. Pair programming: what’s in it for me?. Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. In: ESEM (Empirical Software Engineering and Measurement). ACM, New York, USA, 2008.

[31] Monperrus, M., Jézéquel, J-M., Champeau, J., Hoeltzener, B., 2008. Model-driven Engineering Metric for Real Time Systems. In: Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS’2008).

[32] Arlene Minkiewicz, 2010. Applying Agile Practices to Improve Software Quality, Journal of Software Technology March 2010 Vol.13. Number 1

[33] Tore Dybå, Torgeir Dingsøyr August 2008. Empirical studies of agile software development: A systematic review, Information and Software Technology, Volume 50, Issues 9-10, Pages 833-859

[34] Staron, M., Meding, W., Nilsson, C., 2009. A framework for developing measurement systems and its industrial evaluation. In: Information and Software Technology 51 (2009) 721-737.

[35] Monperrus, M., Jézéquel, J-M., Champeau, J., Hoeltzener, B., 2008. Model-driven Engineering Metric for Real Time Systems. In: Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS’2008).

[36] McQuillan, J., Power, J., 2006. Some observations on the application of software metrics to UML models. In: Workshop on Model Size Metrics as MoDELS/UML 2006.

[37] Lange, C., 2006. Model Size Matters. In: Lecture Notes in Computer Science, 2007, Volume 4364/2007, 211-216

[38] R. Plösch, H. Gruber, A. Hentschel, C. Körner, G. Pomberger, S. Schiffer, M. Saft, S. Storck, "The EMISQ method – expert based evaluation of internal software quality", 31st IEEE Software Engineering Workshop (SEW 2007), pp. 99-108, 2007, doi:10.1109/SEW.2007.71.

[39] Pressman, S., 2005. Software Engineering: A Practitioner's Approach (Sixth, International ed.), McGraw-Hill Education.

[40] Beck, K., et al, "Manifesto for Agile Software Development", Agile Alliance, 14

[41] Mellor, S., J., Clark, A., N., Futagami, T., 2003. Model-driven Development. Software, IEEE, 20(4): 14-18. (Accessed June 13, 2012, from CS Digital Library).

[42] Selic, B., 2003. The Pragmatics of Model-Driven Development. Software, IEEE, 20(5): 19-25. (Accessed June 14, 2012, from IEEE Xplore).

[43] Atkinson, C., Kuhne, T. 2003. Model-driven development: a metamodeling foundation. Software, IEEE, 20(5): 36-41. (Accessed June 14, 2012, from IEEE Xplore).

[44] France, R., Rumpe, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering, 2007. FOSE '07. Minneapolis, MN, USA 23-25 May 2007.

[45] IBM Corporation Software Group, 2009. IBM Rational Rhapsody Developer. [online] Available at: <http://public.dhe.ibm.com/common/ssi/ecm/en/rad14043usen/RAD14043USEN.PDF> [Accessed 14 June 2012].

[46] IBM Corporation, 2009. IBM Rational Rhapsody API Reference Manual. [online] Available at: <http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/rhapsody_api.pdf> [Accessed 14 June 2012].

[47] Bostock, M., 2012. Data-Driven Documents. [online] Available at: <www.d3js.org> [Accessed 14 June 2012].

[48] Wilson, G., & Aranda, J. 2011. Empirical software engineering. American Scientist, 99(6), 466-473.

[49] Gaudin, O., 2009. Evaluate your technical debt with Sonar. SonarSource S.A, [online] Available at: <http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar/> [Accessed 15 June 2012].

[50] McConnel, S., 2004. Code Complete: A Practical Handbook of Software Construction. 2nd ed. Washington: Microsoft Press.

[51] Stavrinoudis, D., Xenos, M., 2008. Comparing internal and external software quality measurements. [pdf] Available at: <http://quality.eap.gr/Publications/XM/Conferences%20English/C53%20-%20Comparing%20Int%20and%20Ext%20Sw%20Q%20Measurements.pdf> [Accessed 24 January 2013]

[52] Monperrus, M., Jézéquel, J-M., Champeau, J., Hoeltzener, B., 2009. Measuring Models. In: Rech, J., Bunse, C., 2008. Model-Driven Software Development: Integrating Quality Assurance. Hershey: IGI Global. pp.170-203.

[53] D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the Impact of Design Flaws on Software Defects. [pdf] Available at: <http://www.inf.usi.ch/phd/dambros/publications/qsic10.pdf> [Accessed 28 January 2013].

[54] Emden, E., Moonen, L., 2002. Java Quality Assurance by Detecting Code Smells. In: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02) (WCRE '02). IEEE Computer Society, Washington, DC, USA.

[55] Radjenović, D., Heričko, M., Torkar, R., Živkovič, A., 2013. Software Fault Prediction Metrics: A Systematic Literature Review. Department of Computer Science and Engineering (GU).

[56] Shah, S., Morisio, M., 2013. Complexity Metrics Significance for Defects: An Empirical View. *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*. London: Springer Berlin Heidelberg. p29-37.

[57] Heijstek, W., Chaudron, M., 2009. Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process. Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference. IEEE, pp. 113-120

Appendix A: Details of Visualizing Tool

This appendix explains different charts and forms shown in the report web page and detailed information.

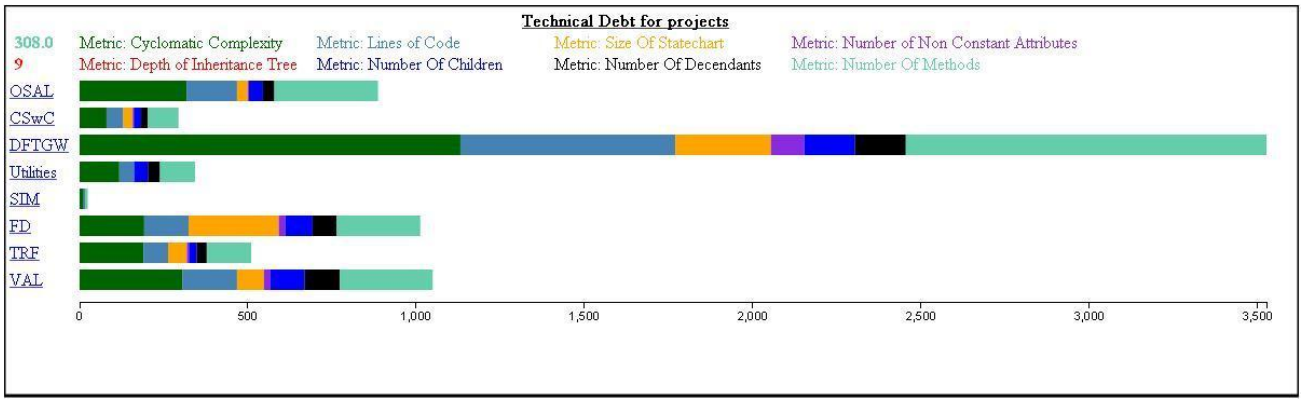

Figure A1: TD for each project

This part is the core chart of the system which shows accumulated technical debt for each project (Part 1 in Figure 5). The bar chart visualizes the total TD shown in different colors based on the amount of TD collected from each metric. For example, from figure A1 it can be seen that the project “DFTGW” has the highest TD out of all the project and the amount of TD in that project comes mostly from violations of the metrics CC, LOC and NOM. Additionally to the TD it also visualizes how many red violations a project has within each metric, this was shown as a red number in the top left corner when holding the mouse-pointer over a specific bar-part.

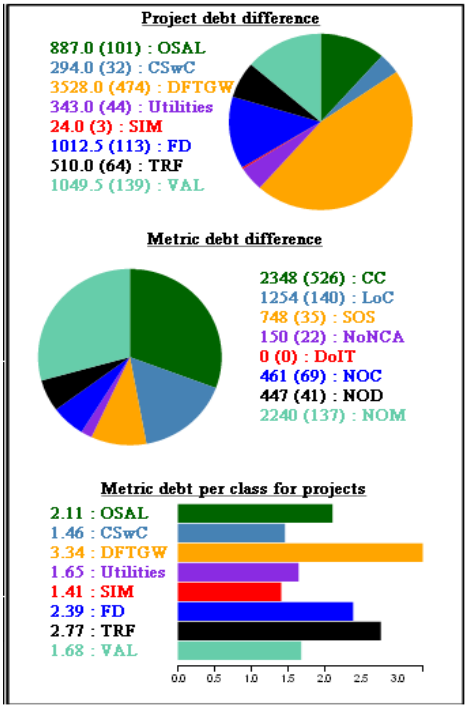

Figure A2: Other TD charts (Part 2 in Figure 5)
The project weight difference (2a) and metric weight difference (2b) was added to visualize the total weight in another shape to comprehend some information easier. Instead of showing the metrics of individual project they show the summed weight of projects compared to each other and the summed weight of the metrics compared to each other in the form of a pie-chart. To each pie-chart a list was added to show which part of the pie-chart was which project/metric and also that parts weight and how many red violations it had in total.

Below the two pie charts is a bar chart which is showing the average TD per class by normalizing the TD for a project by its number of classes. This was created to enable the comparison between different projects while omitting their size.

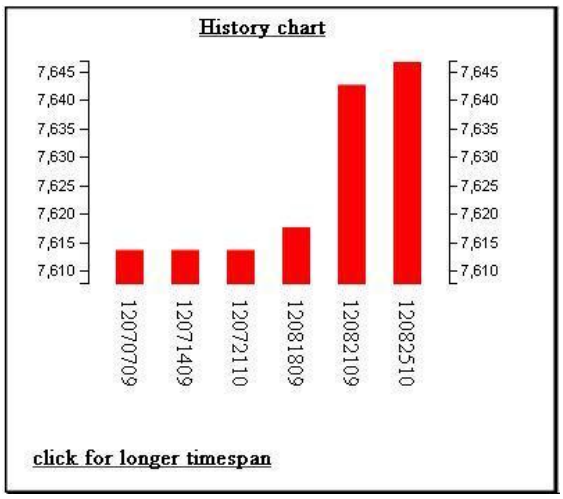

Figure A3: Historical TD Chart Frame
The history chart (Part 3 in Figure 5) is visualizing the total weight of all the projects for up to the last six runs of the MetricAnalyser. Through this chart, the increased/decreased TD would be seen clearly as well as the historical trend. The newest run of MetricAnalyser is the rightmost bar and the oldest one is the leftmost one. The names to each bar in the bar-chart is the date in which the MetricAnalyser has been run. By the wishes from personnel from the development team, a longer timespan was added so that by clicking the link “click for longer timespan”, see figure A3, all historic data will be shown as a broad bar-chart.

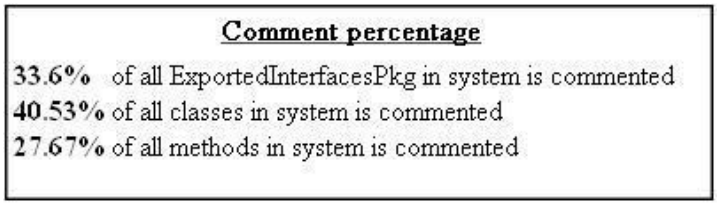

Figure A4: Comment Percentage Frame
Comment percentage (Part 7 in Figure 5) shows in percentage that how many of the classes/methods have a description in the description tag and also all nested packages within the “ExportedInterfacesPkg” package, in all projects, and the “ExportedInterfacesPkg” itself. This selection is the wish of personnel from the development team. The numbers shown in the report are summed together for all system.

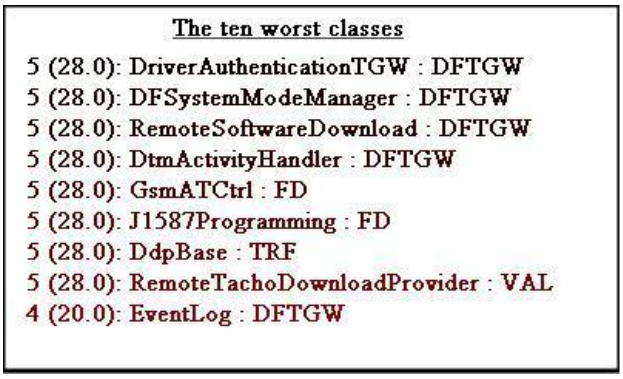

Figure A5: The Worst Classes List Frame

The worst classes (Part 5 of Figure 5) are shown in a list that is ten entries long and was sorted based upon which classes, in all the projects, had the most red violations. The number of red violations was the first, smaller, number that are shown and after that, within the parentheses, were the total weight of the class. Only the name of the class and which project it exist in are visualized in the report visualizing program.

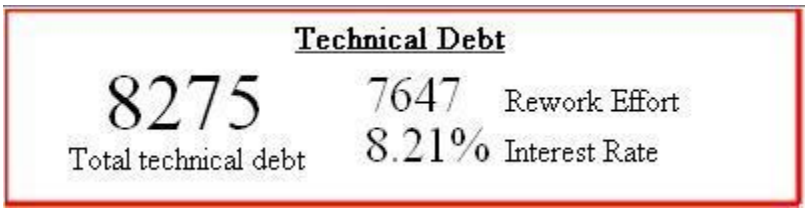

Figure A6: Technical Debt Summary Frame (Part 7 in Figure 5)

The big number is the left shows the total accumulated technical debt (rework effort * (1+ interest)) in man-hours. Rework effort to the right shows the total man-hours needed to fix all the violation exist in those projects. Interest rate to the right is the rate according to which the total technical debt was calculated. The interest rate was calculated by normalizing added time of maintenance related to different level of violations. Detailed interest rate for each level could be found in the configuration file.

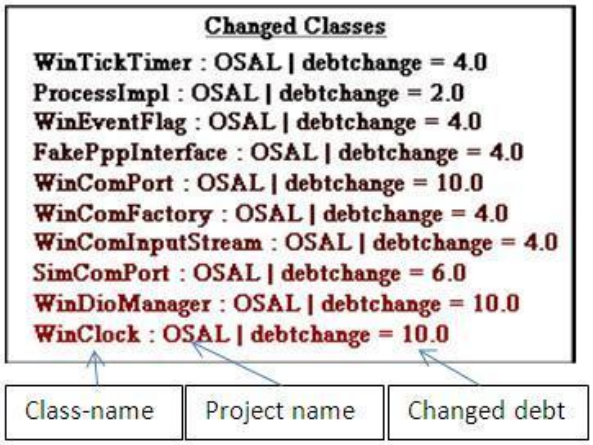

Figure A7: Changed Classes List Frame

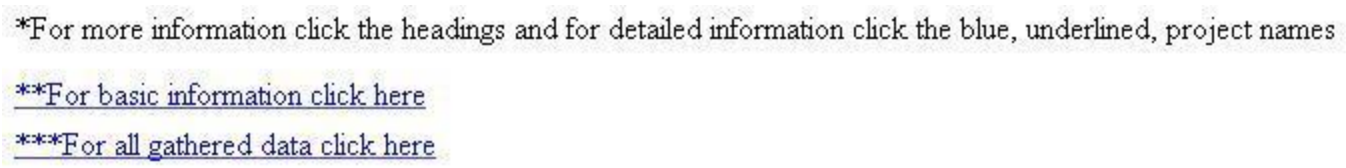

Figure A8: Description and information links (Part 8 in Figure 5)
This part contains information on how to find and reach different parts in the program were listed in the bottom and there also are two links: one that showed all the raw files generated from MetricAnalyser and one link that gave basic information about the metrics, thresholds etcetera.

The basic information page (the first link in Figure A8) was created to remove ambiguity from the report visualizing program and also to show some information which purpose is to ease the use of the report. Some information was from the configuration file, and that information contained the threshold of the metrics and the given weight for the violations. Secondly the abbreviation of metrics were added and given the full name to. Additionally information on what the threshold was and the definition of weight and how that were used was also added. To make it clear how to read the numbers of the metrics the definitions of the metrics done in MetricAnalyser were explained.

By pressing the headlines for any frame/box in the main page, for example “Technical Debt for projects”, an information page will be opened, describing how to read the information in the box with the heading just pressed. All information-pages have a picture and text describing each part of that box, both how they work and how they should be read.

Project Details page:
The project details page included two different things. Firstly it contains a bar-chart (1) that shows the weight for each metric that were accumulated in that project. Secondly it contains one list for each metric (2) with the top ten worst classes in that project, these lists were based on the raw values of the metrics instead of the weight.

Appendix B: Result of first questionnaire

Q1. What is software quality to you?

Fulfills the functionality requirements	16
Is bug-free	13
Code is easy to understand and has clear logic	13
Is easy to maintain	13
Is efficient and fast, no memory leak, consumes little resources (i.e. space, memory, CPU time etc.)	9
Everything is well documented	5
Fulfills customer needs	2

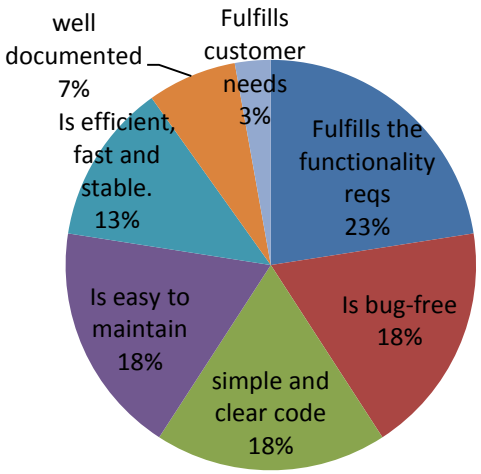

Chart 1: Pie chart of answers for question 1

Q2. How would you measure software quality if you could decide?

Bug Number	20
Unit test coverage, test coverage	19
Complexity of code logic, connections between classes	15
Lines of code per function	1
Well-defined interfaces	1
Number or relevant comments	1
Profiling	1
Solved bugs that still not pass the retesting phase	1

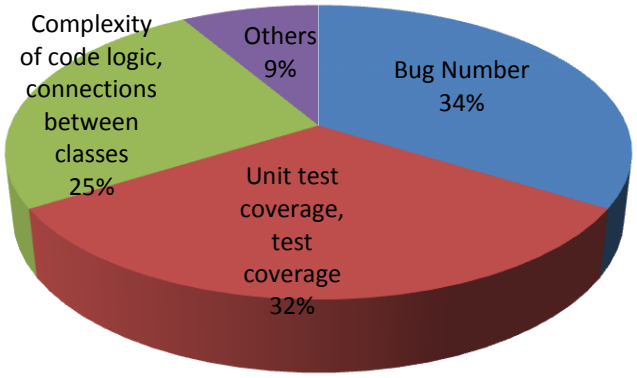

Chart 2: Pie chart of answers for question 2

Q3. What do you think about the current product quality?

3 trucks	4	17%
4 trucks	1	4%
5 trucks	8	35%
6 trucks	6	26%
7 trucks	2	9%
8 trucks	2	9%

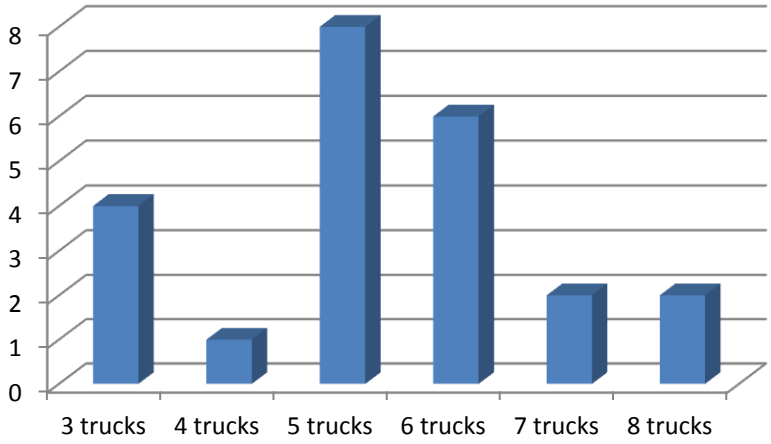

Chart 3: Ranking of product quality from team members

Q4. How do you think we can improve the product quality?

The keywords mentioned in the survey was abstracted and categorized into 5 categories.

Category	Sum	Subject	Numbers
General/Team/Environment/HW	5	Improve physical working environment	1
		Better machine (X64)	1
		Better programmer	1

		Better team work	1
		Open climate	1
Development	24	Better unit test	4
		Better code coverage	4
		More best practice	3
		Easier for developer to test	2
		Easier unit test	2
		Understanding impact of (change) each function	2
		Guidelines for unit test	1
		Awareness of quality from developers	1
		Pair programming	1
		Work rotation	1
		Better CM strategy	1
		Rebase often	1
		TML code more modular	1
Testing	9	More test	2
		Better test	2
		More fault detection (memory leak, deadlock)	1
		More target testing	1
		More integration test	1
		Test earlier	1
		Test requirements and design	1
Process	15	More reviews	6
		Wider range of review	1
		More strict process	1
		More check point	1
		More focus on non-functional requirements	1
		Meeting &information for best practice	1
		Easy solutions to functions	1
		Simple architecture	1
		Rewrite part code	1
		More time on design	1
Tools	12	Remove Rhapsody	4
		Better build process	3
		Use tools more	2
		Better static checking tools	1
		Check-in threshold	1
		More metrics	1

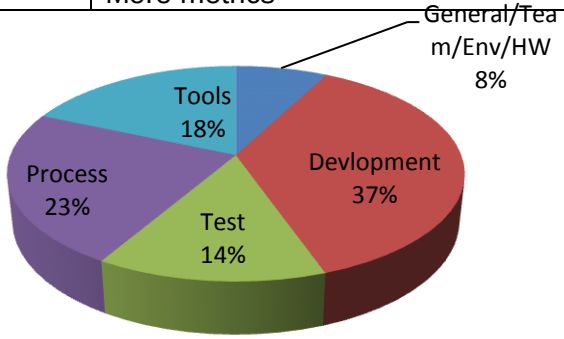

Chart 4: Percentage of categories for keywords mentioned in answers for improvement for Question 4

Q5. How do you think we can improve our development process?

Here the keywords mentioned in the answers are abstracted divided into three categories.

Category	Sum	Subject/Keywords	Numbers
Development Process	26	Shorter Cycle	4
		More documents	4
		Clear dev process	3
		Earlier testing	2
		Automated regression test	2
		Integration early	1
		Agile	1
		Strict process	1
		Faster build process	1
		Use DevTrack process	1
		Test Driven Development	1
		Unit Test focus on method/small function	1
		Focus on coding, less meeting	1
		Better requirements	1
		Code review	1
Development practice	19	Share responsibility and pair programming	3
		Best practices	3
		Easier for developer to test	2
		Better understanding of functions (use scenarios)	2
		Knowledge sharing	2
		Unified coding style	1
		Double check from developers	1
		Rhapsody training	1
		Templates and examples	1
		Early release of test programs	1
		Replace Rhapsody, use java, C#, C++ instead	1
Organization and team	2	Flatter organization	1
		Smaller teams focus on specific area	1

Appendix C: Result of workshop

Time: 3rd of April (10:00 – 12:00), 4th of April (10:00 – 12:00)
Participants: Fredrik, Kristian K, Mattias, Christian J
Facilitator: Björn, Sean

Workshop I

Metrics (Group I):

1. Prioritization (start with the easies first):
    I. Lines of code:
    II. Cyclomatic Complexity
    III. Size
    IV. Comment Percentage
    V. The rest
2. Scales, explanations of metrics
    I. Lines of code
        i. Green < 80
        ii. Yellow 81- 199
        iii. Red > 200
        iv. Lines of code should be calculated on the implementation tabs code. (2 pages are too much.
    II. Cyclomatic complexity
        i. Green <= 3
        ii. Yellow 4-5
        iii. Red > 5
        iv. Should be calculated on the implementation tabs code.
    III. Size (Model metrics)
        i. Number of methods
        ii. State-charts
            1. Number of states
            2. Number of transitions
        iii. Number of non-constant attributes
    IV. Comment Percentage
        i. Should not be over 30% in implementation tab
        ii. On interfaces (as description) it should be 100% rest is red
    V. Weighted Method per Class (Model metric)
        i. Should be combined with depth of inheritance tree to make a sum of how bad it is because that gives a better number
    VI. Coupling Between Object classes (Model metric)
        i. Have to differentiate between interface and implementation

   ii. Have to measure for both too high and too low values because too low is no good either

VII. Response For a Class (NOT TO BE IMPLEMENTED)
   i. Shouldn't be implemented, it is too hard to measure and implement

VIII. Lack of Cohesion in Methods (NOT TO BE IMPLEMENTED)
   i. Same as for RFC

IX. Depth of Inheritance Tree (Model metric)
   i. Have to differentiate between interfaces and implementation
   ii. Green < 3
   iii. Red > 5
   iv. Good to combine with WMC

X. Number Of Children (Model metric)
   i. Good to have but should rather warn that it is shouldn't be changed when the number is higher, other than that it is ok if the number is high.

3. Other:
   I. Must be a possibility to suppress classes that shouldn't be measured.
   II. Warning should be for at least as high as class so that "we" can go deeper into it afterwards, otherwise there will be too much information (there is thousands of classes)

Metrics (Group II):

Prioritized List:

1. Cyclomatic Complexity
2. Lines Of Code, size
3. Comment Percentage

LOC: In different function level: Application, module/package, class, function

Size: compare historical information could be useful

CC: Green 10, Yellow ?, Red ? (Depend on the average result from current code, not in generated code but inside Rhapsody)

CP: depend on what kind of code, Green 30% (implementation, 100% for interface, not required on simple functions) Good to have for information.

CBO: a bit complex to implement, also good to have for information. Cross reference is very interesting, if can detect that, would be very helpful.

DIT: the number is good in current code, good to have for information.

NOC: instead of children, number of descendants (including children's chidren) is more interesting.

LCOM: good to have, but complex to understand, hard to implement.

Some metrics should be combined together, e.g. DIT & WMC.

Conclusion of discussion on metrics:

First implement metrics:

1. Lines of code
2. Code complexity
3. Size
    a. Number of methods
    b. State-charts
        i. Number of states
        ii. Number of transitions
    c. Number of non-constant attributes
4. Comment Percentage

Continue with the other metrics if there is time (report of metrics is more important). Some of them are good to have but not very intuitive, also not very feasible to implement.

Coding Standards:

- "Metrics is one way to enforcing coding standards" – Mattias
- We (Björn & Sean) can look on what research (state-of-the-art) say about code review
    - Give suggestions from this
- Look into peer review (state-of-the-art)
    - Any tools for doing it
- Quality gates are not feasible at check-ins, may work to have them when it is time to ship the product.

Workshop II

Unit Tests & Coverage:

Unit test coverage would be hard to introduce a standard for since the current coverage is really low and setting a low standard is not very useful either. One option is to start at a lower standard and increase it. Problem is that to have a coverage threshold the developers have to work with coverage in mind which is not what they are doing now.
Today there are only ambitions, "One ambition is to have at least one UT for each module."
If there is any measurement on standards, it should only consider newly added code/modules instead of all the code/modules.
According to workshop team it would be good to get numbers on how much coverage other companies have and what research say about UT & coverage.

Technical Debt:

There was a really long discussion. People in the workshop think TD is really interesting but might not feasible. The thing that can be done is to make a base to continue working on, see summary of workshops for what we are going to work on.
The main concerns are:

1. There is no role that is responsible for the TD for a project.
2. No existing standards to calculate the TD.

3. Hard to define how long it takes to fix a problem, would have to be very rough.

There should be different weights for different measures (metrics, coverage and so on) if the TD would be calculated. Another thing to take in mind is that things can/will not be done with old work and therefor will give of a huge debt that will not be fixed. This will result in that new debt that is added will not be seen as important as it otherwise would.
We didn't manage to reach an agreement, more input are needed and further discussion will be carried out towards this.

Other topics

We also discussed agile process, knowledge sharing and other metrics (process metrics, organizational metrics) just for informing the team about the possibilities and methods/best practices to improve code quality.

Summary of Workshops and what to do next

Start working with metrics first, they shall be implemented by java-scripts in Rhapsody. It is important to get the numbers so that there is some information to work with. The metrics we should implement are the metrics that were discussed in the first workshop. When those numbers and statistics are in the systems the improvement group/workshop group can work on setting standards and new goals towards better code quality.

The metric scripts shall be configurable, in order to ignore some known issues. This shall make it possible to exclude code and modules that is not wanted.

All the metrics shall be calculated to one number so that there is a general "quality number" that for example is showing that the quality trend (as well as details) of past day/week is good or bad in a single number/line of text /table/chart.

The result collected from metrics shall be calculated in a non-accumulated way. For example, start from 0 every day/week, to show the impact of newly added code or modified code to the code quality.

The result shall be shown in a daily/weekly report which is sent to related alias. And all the reports shall be customized according to the receivers.

Then the scripts should be integrated in the nightly build so that the results can be reported every day. But the scripts should also have the possibility to be run manually whenever it is needed and then on any parts wanted.