TUCS

Jeanette Heidenberg | Jussi Katajala | Iván Porres

Maintainability Index for Decision Support on Refactoring

Turku Centre *for* Computer Science

TUCS Technical Report
No 992, November 2010

Maintainability Index for Decision Support on Refactoring

Jeanette Heidenberg
TUCS Turku Centre for Computer Science
Åbo Akademi University
Joukahaisenkatu 3-5, FI-20520 Turku, Finland
jeanette.heidenberg@abo.fi
Jussi Katajala
OY LM Ericsson Ab
Hirsalantie 11, FI-02420 Jorvas, Finland
jussi.katajala@ericsson.com
Iván Porres
TUCS Turku Centre for Computer Science
Åbo Akademi University
Joukahaisenkatu 3-5, FI-20520 Turku, Finland
ivan.porres@abo.fi

TUCS Technical Report

No 992, November 2010

Abstract

Maintainability is a software attribute that needs to be continuously addressed during the entire development life-cycle. But the decision to refactor in order to keep the software product maintainable is not always an easy one. In this paper, we present a metrics-based approach for assessing the maintainability of code under development with the purpose of providing decision support for refactoring decisions. Our approach is developed and validated in the context of a large, mature telecommunications product.

Keywords: software maintenance, software quality

TUCS Laboratory
Software Engineering Lab

1 Introduction

According to the IEEE standard glossary [1] maintenance is “the process of modifying a software system or component after delivery.” As such, maintenance is usually associated with the last stages of the development cycle. This may not necessarily be the case in an iterative development process such as the Rational Unified Process [16] or Scrum [22], where the software is effectively under maintenance already after the first iteration. The cost of maintaining and further developing a poorly designed system can be high. This cost increases the longer the technological shortcomings go uncorrected, until the code becomes unmanageable and essentially has to be completely rewritten. This escalating problem is often called technical debt, or design debt [7]. In order to ensure that the technical debt does not escalate out of control, the maintainability of the product under development should be addressed in all stages of the software process.

Refactoring [11] is the state of the practice for fighting the build-up of technical debt. The design and architecture of the product should be under constant improvement in order to accommodate new constraints and requirements. Looking for code and design “smells” (code or design patterns indicating bad design) and amending these should be an integral part of the development work.

In reality the development team often has a difficult choice to make. Whereas smaller refactoring efforts can usually be easily included in the implementation effort, larger changes may require a greater effort and have to be planned for. The project will face situations where they have to weigh the cost of refactoring against the cost of cutting features from the delivery. This decision is essentially a business decision, but it cannot be made without a clear insight into its technical merits. In a large corporation the business expertise and technical expertise is usually represented by different people, often in different organizations, working towards different goals and using different vocabularies. A lack of trust between the two is not uncommon. For these reasons, reaching a decision to refactor may not be a trivial task.

In this paper, we present a metrics-based approach for assessing the maintainability of code under development with the purpose of providing decision support for refactoring decisions. Our goal is to provide development teams with a method for assessing and visualizing the technical debt in a way that supports decision making and facilitates prioritization of refactoring efforts.

Our approach is developed in the context of a large, mature telecommunications product. The product in question has been under development for several years and has seen a development effort comprising hundreds of person years, using several programming languages. Metrics from a number of subsystems developed using IBM Rational Rose RealTime are collected. A model for quantifying the maintainability of the subsystems is created and calibrated against the technical experts' intuition of the maintainability of the subsystems in question. We also validate the metrics against the before and after states of a large refactoring effort (of one person year) known to have had a positive effect of maintainability.

2 Background

2.1 Software Metrics

Software metrics are usually classified [9] as either product metrics or process metrics. A *software product metric* is a function that quantifies a property of the measured software. Lines of code (LOC) is an example of a software product metric. In contrast, a *software process metric* is a function that quantifies a property of the process used to develop software. Average LOC per person-month is an example of a process metric.

In this paper we suggest the usage of metrics in order to assess the maintainability of code under development. The purpose of this assessment is to provide the project with decision support for refactoring. We believe that a good collection of trusted metrics can facilitate communication between business stake-holders and technical experts to this end. We issue a strong warning against the temptation to use these metrics to draw conclusions regarding the skill or professionalism of the developers. It is essential to understand that a subsystem's maintainability is not necessarily a reflection of the competence of the development team. Many other factors influence this attribute. These include the complexity of the implemented features, the maturity of the product itself, the maturity of the used interfaces, clarity of the requirements, and time pressure to mention but a few. For this reason, the metrics proposed here should be used as information measurements only and never for comparing the "goodness" of teams or individual developers.

2.2 Measuring Maintainability

We strive to measure the maintainability of software. Intuitively, we define maintainability as the ease of which a software system or component can be modified to add or remove features, correct defects, improve quality attributes such as performance, or adapt to changes in its environment.

We base our definition on standards such as the ISO 9126 [14]. Maintainability is one of the six quality attributes listed in the ISO 9126 standard, alongside functionality, reliability, usability, efficiency, and portability. As we are addressing maintainability from the point of view of the developer, we are interested in the internal aspects of this attribute, as opposed to the external ones observed by, e.g., the customer. More specifically, we are interested in the sub-attributes analyzability and changeability.

In [12] we demonstrated by means of analysis of historical data as well as a controlled experiment, that the use of certain design constructs can indicate low maintainability in Rational Rose RT models. Briand et. al. have reached similar conclusions [4] for object oriented software. Studies provide sets of OO metrics for Java [17] and C++ [5] that can be used for decision support for refactoring. Mäntylä and Lassenius show that such metrics can indeed be used to predict refactoring needs [18], especially if complemented by manual reviews of critical parts of the code [19]. It is even possible to estimate the cost of maintenance based on complexity metrics, as demonstrated by Fioravanti [10]. In contrast, Yu et. al. argue that it is not possible to measure maintainability. However, their results make use of process metrics only and are restricted to open-source contexts.

Coleman et.al. [6] present a number of methods for deriving a maintainability index based on internal product metrics. We loosely follow their approach, especially with regards to taking advantage of the intuition of the developers in order to calibrate their index.

Moha et.al. [20] present a complete method for identifying code and design smells, starting from the identification and definition of smells, continuing with processing the definition into executable algorithms, actual detection of smells, and ending with the manual validation of the found smells.

2.3 M-MGw Software

This study was performed in the context of the Ericsson Media Gateway for Mobile Networks (M-MGw) [8], which is a part of the softswitch solution for Ericsson's Mobile Core Network. The M-MGw was first commercially launched in early 2003 and has since then been under development and maintenance with more than 100 people working actively in the projects at any one time. The M-MGw is a mature and industrialized product, with an install base of several hundred units in customer networks around the world.

The M-MGw is developed using 3 different languages and environments: C, Java, and C++ in IBM Rational Rose RealTime. C is used to implement low-level code running on digital signal processors (DSP). The C code includes strict resource and real-time constraints. Java is used to implement monitoring and configuration facilities, including their user interfaces. The Java code has no special resource or performance constraints. IBM Rational Rose RealTime is used to describe reactive software using statecharts and C++. The Rational Rose RealTime models contain performance constraints, but no real-time constraints. The models contain action code in C++ and are compiled to executable software with no separate step where the generated code would be modified. This study was performed in the context of the subsystems developed using C++ in IBM Rational Rose RealTime.

2.4 IBM Rational Rose RealTime

IBM Rational Rose RealTime is a modelling tool for reactive systems, used for model-driven development with a subset of the UML 2.0 [21] standard called UM RealTime.

The Rational Rose RealTime modeling approach is based on the concept of a capsule. A capsule is an active component with its own thread of control. It can communicate with other capsules via ports. A port may require or realize an interface and an interface is defined as a set of signals. Capsule communication is asynchronous. The main behavior of a capsule is specified using a statechart diagram. In the M-MGw product, the guards and actions of a statechart are defined using the C++ programming language. A capsule may contain other capsules and passive objects, which are instances of C++ classes.

The execution of a transition is triggered by the reception of a signal on one of the capsule's ports. The triggered transition decides whether it deals with the signal immediately or defers it until later. A deferred signal may be recalled at any time during execution. When a transition is triggered, the currently active state may change. If the previously active state contained an exit action, the code within this action will be executed before the transition code is executed. If the target state contains an entry action, the code within this action will be executed after the transition code is executed. Transitions may be composed of multiple transitions, either through nesting of states or by choice points. A choice point contains a boolean expression, and depending on the value of this expression the next transition in the chain is selected.

2.5 The Challenges of Technical Debt

This work is part of a larger body of work instigated by the studied organisation as the answer to a specific need. As the model driven approach of IBM Rational Rose RealTime was rather new to the industry at the point in time of the study, there was a lack of consensus among the developers as to what constitutes good statechart design heuristics.

Partly as a consequence of this lack of consensus, the organisation did not properly address technical debt. The business stakeholders and technical staff did not have a good way of discussing technical debt. The business stakeholders suspected the developers of being overzealous in perfecting the technical details of the system, while the technical staff felt they were not given enough time to restructure and refactor code that had deteriorated over time.

The technical debt of one subsystem eventually reached a limit where it could not be properly maintained any longer. A major restructuring effort was carried out with good results, and the organisation decided to learn more about maintainability of IBM Rational Rose RealTime models and how to, properly and in a timely fashion, address technical debt.

The question of what constitutes good statechart design was studied using statistical methods and a controlled experiment and was reported in [12]. The question of how to address technical debt is discussed in this paper.

3 Towards a Maintainability Index

This study was performed in a specific context with a specific goal of improving the practices of the researched organization. This mainly affects the study in the fact that the approach is pragmatic. The metrics need to be easy to collect, in order for the organization to be able to deploy this practice without encumbering the daily work of the developers. The metrics presentation should be easy to understand, in order for it to serve as a good decision support system for both technical and business staff. The metrics should be trustworthy, in order for it to be effective as a negotiation tool.

In order to serve the pragmatic needs of the organization, the metrics collected are summarized in a *maintainability index (MI)*. The purpose of the maintainability index is to give an estimate of the technical debt of the system in order to provide the projects with decision support on when and where to intensify the refactoring efforts.

The M-MGw product is implemented in an incremental, iterative manner. Small, internal deliveries are built and delivered for internal quality assurance following a regular cadence. A number of such internal deliveries constitute a formal delivery of the product. Quality assurance is a continuous effort on different integration levels and a significant effort is invested in ensuring that the formal delivery is of high quality.

In building a model of the maintainability of the IBM Rational Rose RealTime models of the system under study, we collected metrics from seven subsystems of one delivery of the M-MGw product. This we calibrate against the intuition of subsystem experts regarding the maintainability of the subsystems in question. In the following, we account for the way these data were collected and how they were used to evolve a model. This model was finally validated by checking it against three deliveries of the same subsystem: a version known to have poor maintainability, a restructured version known to have significantly improved maintainability, and the version which was current at the time of the study, and which was known to have deteriorated somewhat since the restructuring effort.

	Sys1	Sys2	Sys3	Sys4	Sys5	Sys6	Sys7
Rating:	M	H	L	H	H	M	L

Table 1: Maintainability Rating per Subsystem

3.1 Expert Intuition

For model calibration, we used the intuitive perception of a group of experts regarding the maintainability of seven subsystems of the embedded system developed by the organization in question. These experts were seasoned developers and architects with extensive experience in working with the specific subsystems. They were either the technical owner of the subsystem in question or held the role of software design architect or system architect.

This group of experts had formed with the objective to evaluate the quality of the architecture of one specifically problematic subsystem. During this effort, one of the conclusions they reached was that they did not always agree on what constitutes good design and whether a certain solution would be a smell or an acceptable solution. This is an issue that is somewhat overlooked in approaches such as the one proposed by Moha et. al. [20], but that in our experience is not uncommon in the industry.

What the experts did agree on, however, was the relative maintainablitiy of the subsystems, when compared to each other. This was based on their vast experience in working with the different subsystems, systematic architectural reviews of the systems as well as experience of the defect trends in the subsystems over serveral years. Table 1 depicts this intuitive evaluation of the maintainability of the subsystems on a a three-point ordinal scale, where one (L) signifies low maintainability and a clear need for refactoring, two (M) signifies medium maintainability and possible need for refactoring and three (H) signifies high maintainability and no need for refactoring.

It is worth noting that the evaluation was not explicitly collected for this study, but was rather a by-product of the architectural evaluation effort. Given more time and resources, we would have asked the experts to make a more detailed rating. As we were limited in this respect, we chose to use three-point scale, since a more detailed scale would have required a larger evaluation effort by the experts.

3.2 Metrics Collection

The data points were gathered from three internal deliveries of the Rational Rose RealTime subsystems of the product. First, we looked at the (then) current internal delivery which would best match the experts' intuition. We refer to this delivery as d_n. We also looked at historical data for one of the subsystems, $Sys6$. This subsystem underwent major restructuring at an earlier point in the lifetime of the product, since it had been determined that its technical debt had lead to an increasing maintenance effort. We look at the delivery before (d_{r-1}) and just after (d_r) the refactoring of this subsystem.

Ericsson as a company values process maturity and as such has a high standard of adherence to good software development practices. Good tools and practices for configuration management are used. The software repository contains ample information on what version of the code was delivered in which delivery, thus enabling us to easily extract the relevant data from the repository.

As there were no commercial tools available for collecting metrics on systems designed using IBM Rational Rose RealTime, we collected metrics using an in-house tool designed specifically for the purpose.

As a part of the earlier effort to clarify what constitutes good design of statecharts [12] we had a studied a vast collection of metrics, and ended up with a list of problematic design idioms to look for. Based on this study, we also have a set of threshold values (or "trigger points" according to the terminology used by Coleman et.al. [6]) for when these design idioms should raise a warning flag. Furthermore, we used simple size and complexity metrics to evaluate the maintainability of the action code in transitions, methods and choice points.

Below follows a list of the selected metrics. Some of the selected metrics are related purely to the visual aspects (V) of the state machines, while others are related to the action code (C) in C++. The thresholds values were selected based on the experience of our expert group, and were appropriate for the context in question at the time of the study. The intention is for the thresholds to be evaluated and updated periodically to reflect the current state of the system as it evolves.

(*V*) Ratio Choice Points / State This metric calculates the ratio of choice points per state in the statechart. A large number indicates that there are many different paths through the statechart. This may indicate the antipattern commonly known as "flag jungle" among the organisation's developers. In this antipattern, the statechart depends on too many attributes for decision making and some form of abstraction should be considered, either by splitting the offending capsule or by turning some of the attributes into states. The suggested threshold for this metric is 1.

(*V*) Visual Cyclomatic Complexity This metric calculates the cyclomatic complexity of the statechart as the number of transitions plus two minus the sum of the number of states and choice points: $\sum transitions - (\sum states + \sum choicepoints) + 2$. The suggested threshold for this metric is 100.

(*V*) Visual Size This metric calculates the number of visual elements in the statechart (ports, operations, attributes, states, choice points, transitions.) The suggested threshold for this metric is 300.

(*C*) **Method and Transition LOC** This metric calculates the number of efficient lines of code in methods and transitions. The suggested threshold for this metric is 80.

(*C*) **Method and Transition Cyclomatic Complexity** This metric calculates McCabe's cyclomatic complexity for action code in methods and transitions. The suggested threshold for this metric is 10.

(*C*) **Choice Point LOC** This metric calculates the number of efficient lines of code in choice points. The suggested threshold for this metric is 20. Note that this is much lower than the threshold for methods and transitions. This is due to the findings in our previous study, where choice points seem to have a higher impact on the defect rate.

(*C*) **Choice Point Cyclomatic Complexity** This metric calculates McCabe's cyclomatic complexity for action code in choice points. The suggested threshold fro this metric is 5. Please note that this is much lower than the threshold for methods and transitions, for the same reason as the LOC metric.

(*C*) **Entry / Exit Actions** This metric indicates whether there are entry or exit actions in the statechart. (Warning)

(*C*) **Defer / Recall** This metric indicates whether there are calls to defer or recall in the action code of the statechart. (Warning)

We use the term *design flaw* to denote a location in a subsystem where the measurement for one of the metrics listed above falls outside of the trigger. We do not attempt to measure the degree of fit, i.e., much the design flaw deviates from the trigger point range. Please note that we do not consider a design flaw to be a defect. Although a design flaw may make the subsystem more difficult to maintain, it does not necessarily degrade the functionality of the subsystem. We also make this distinction for the reason that, although the aim usually is to deliver a product without defects, it is not always economically feasible to deliver a flawless product.

3.3 Definition of a Maintainability Index (MI)

Based on the metrics collected and with the calibration the intuition of the developers we proceed to construct a model for assessing the maintainability. We call this model a maintainability index (MI). We divide our MI into two parts: the MI for action code (MI$_c$)and the MI for visual elements (MI$_v$).

Our MI is based on the total number of design flaws, both visual and code based. The count of design flaws found for the visual part of the measured subsystem is denoted fl_v. The count of design flaws found for the action code is denoted fl_c.

The maintainability index is not normalized over size, as the intention is first and foremost to help the developers understand the size of the technical debt by finding and flagging indications of problematic design. Normalization over size would hide the true extent of the technical debt in large subsystems.

It may be difficult to understand the maintainability of a subsystem given just on a number, though. If a subsystem has the a visual MI of 46 and code MI of 2, should that subsystem be improved or not? In order to simplify evaluation, the maintainability index is translated to a number between 0 and 1, where 1 indicates very good maintainability and 0 indicates very poor maintainability.

The first step in this translation is to consider the relative weight of the two parts: the code flaws (fl_c) and visual flaws (fl_v). Looking at the collected data from delivery d_n, we observe that the difference is in an order of magnitude of 10 (see Figure 1). Further evaluation places the median for the ratio fl_c/fl_v at about 20 (see Figure 2. The outlier is the reworked $Sys6$, which had exceptionally few visual flaws due to the improvement work it had seen some deliveries earlier.

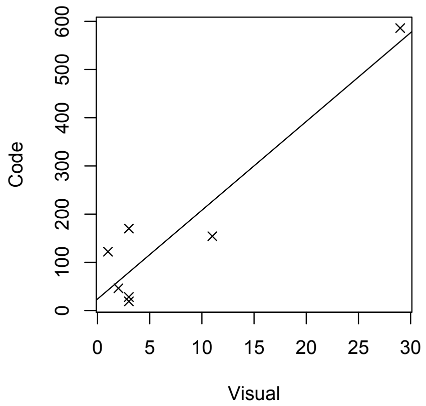


Figure 1: The number of code flaws and visual flaws for each subsystem.

We continue to plotting the subsystems of delivery d_n together with the expert rating of the subsystems. The rating is translated to numeric values between 0 and 1, so that low maintainability (L) corresponds to value 0, medium (M) to value 0.5 and high (H) to value 1. When looking at the

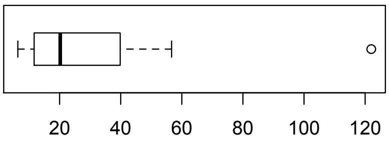


Figure 2: Boxplot of code flaws divided by visual flaws.

resulting graph (see Figure 3) , we notice that $Sys3$ seems to be an outlier. This was known to be an especially problematic subsystem, and the experience was that the maintainability of this subsystem was much lower than that of the others. The figure also shows a linear fit of the subsystems, excluding outlier $Sys3$.

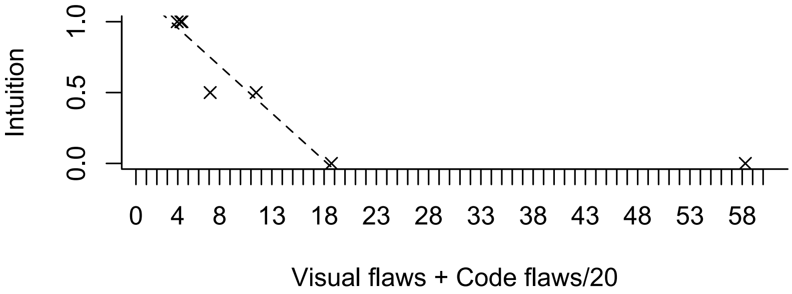


Figure 3: A plot of $\text{fl}_v + \text{fl}_v/20$ of the sybsystems of delivery d_n.

Using this linear fit, we define the overall maintainability index for our IBM Rational Rose RealTime subsystems. The index is limited to the interval [0..1] and defined a follows:

$$\text{MI}(\text{fl}_c, \text{fl}_v) = min(max(\frac{19}{16} - \frac{\text{fl}_c/20 + \text{fl}_v}{16}, 0), 1)$$

Figure 4 depicts this function. In order to make the maintainability index more easily accessible for non-technical staff, we define five color coded levels ranging from red (for very poor maintainability) to green (for very good maintainability). The thresholds for these five levels can be seen in the figure and were derived by dividing the interval [3..19] (representing the minimum and maximum of the function) in equally sized sections.

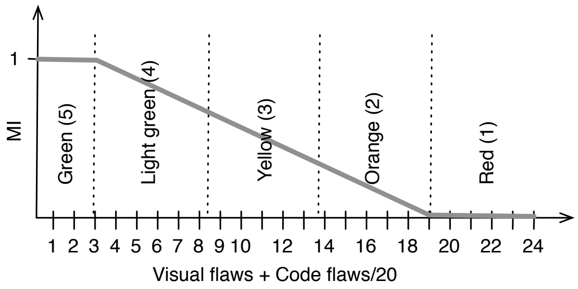


Figure 4: Maintainability function for IBM Rational Rose RealTime Models

For transparency reasons, MI can be split up into its components (MI_c and MI_c). This way it is easier to see whether a subsystem needs code improvements or improvements in the visual elements. Using the fact that we have constructed our MI as $\mathrm{MI}_v + \mathrm{MI}_c/20$ we deduce the following two functions for MI_c and MI_v, where fl_v and fl_c are the number of design flaws for visual elements and action code respectively.

$$\mathrm{MI}_c(\mathrm{fl}_c) = min(max(\frac{19}{16} - \frac{\mathrm{fl}_c}{160}, 0), 1)$$

$$\mathrm{MI}_v(\mathrm{fl}_v) = min(max(\frac{19}{16} - \frac{\mathrm{fl}_v}{8}, 0), 1)$$

Figures 5 and 6 depict these functions.

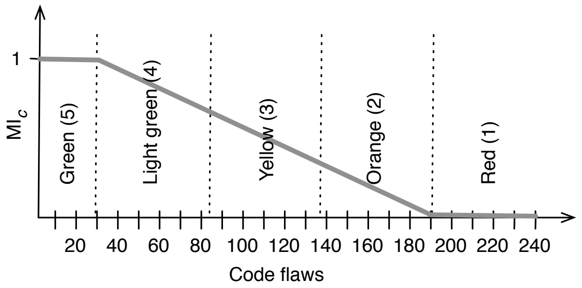


Figure 5: Maintainability function for code flaws.

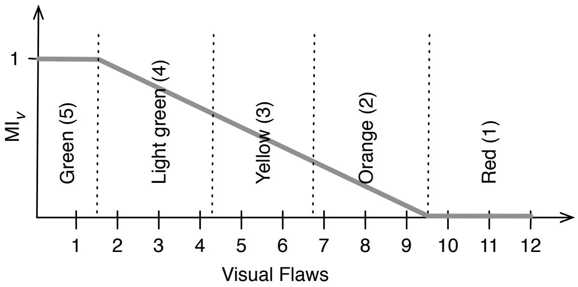


Figure 6: Maintainability function for visual flaws

3.4 Data Presentation and Evaluation

Table 2 lists the metrics gathered for delivery d_n of each of the subsystems, and also for deliveries d_{r-1} and d_r in the case of subsystem $Sys6$. The table also shows the calculated values for MI$_c$, MI$_v$ and MI$_{rrt}$ together with their corresponding color levels, as well as the expert rating for each of the subsystems.

3.5 Validation against a Refactoring Effort

Figure 7 highlights the different values for MI for the three deliveries of $Sys6$. The first (d_{r-1}) is the last delivery before the refactoring; the second (d_r) is the first delivery after the refactoring; the third (d_n) is the current delivery. As we can see, the maintainability index increases substantially after the refactoring. During this refactoring, both the code and the visual design were improved. As a result, the project could measure a significant decrease in defect reports and many of the defects that were found in earlier deliveries and mapped to the current delivery could be discarded, since the fault just simply did not exist any more due to the refactoring. The size of this refactoring effort was approximately one person year.

It is also interesting to note that $Sys6$ has seen a slight degradation in maintainability between d_r and d_n, when no refactoring has been done. The experts have noticed this degradation, and we can also measure a lower maintainability index.

4 Suggestions for Deployment Practices

In this section we give some suggestions on how to deploy a software maintainability index in an organization.

We suggest that the MI metrics should be collected at least at delivery and the resulting list should be stored. For every warning, an action should

	Sys1	Sys2	Sys3	Sys4	Sys5	Sys6	Sys6	Sys6	Sys7
	d_n	d_n	d_n	d_n	d_n	d_{r-1}	d_r	d_n	d_n
Visual size warnings:	0	0	3	0	2	0	0	0	0
Visual cyclomatic complexity warnings:	1	2	4	0	1	2	0	1	2
Visual choice points/states warnings:	2	1	4	0	0	0	0	0	5
Number of entry actions (warnings):	0	0	11	1	1	3	2	0	4
Number of exit actions (warnings):	0	0	7	2	0	3	0	0	0
Class operation amount warnings:	3	4	2	0	1	2	2	2	6
Number of defers (warnings):	15	0	48	8	3	45	19	30	28
Number of recalls (warnings):	4	0	19	7	4	32	14	20	23
Class operation LOC warnings:	27	1	49	0	5	2	5	14	16
Capsule operation LOC warnings:	11	1	44	2	0	0	0	1	5
Transition LOC warnings:	19	6	28	2	11	2	2	6	3
Choice point LOC warnings:	0	3	36	1	0	0	0	0	1
Class operation CC warnings:	38	0	160	0	12	5	17	29	42
Capsule operation CC warnings:	26	1	96	4	0	11	1	9	18
Transition CC warnings:	27	1	60	3	10	11	5	11	11
Choice point CC warnings:	0	2	44	1	0	0	0	0	1
SUMMARY:									
Total number of code warnings:	170	19	586	28	46	110	65	122	154
Total number of visual warnings:	3	3	29	3	2	10	2	1	11
MI$_c$:	0.13	1.00	0.00	1.00	0.90	0.50	0.78	0.43	0.22
Color level:	(2)	(5)	(1)	(5)	(4)	(3)	(4)	(3)	(2)
MI$_v$:	0.81	0.81	0.00	0.81	0.94	0.00	0.94	1.00	0.00
Color level:	(4)	(4)	(1)	(4)	(4)	(1)	(4)	(5)	(1)
MI:	0.47	0.94	0.00	0.91	0.92	0.21	0.85	0.74	0.02
Color level:	(3)	(4)	(1)	(4)	(4)	(1)	(4)	(4)	(2)
Expert rating:	M	H	L	H	H	L	H	M	L

Table 2: Metrics per Subsystem

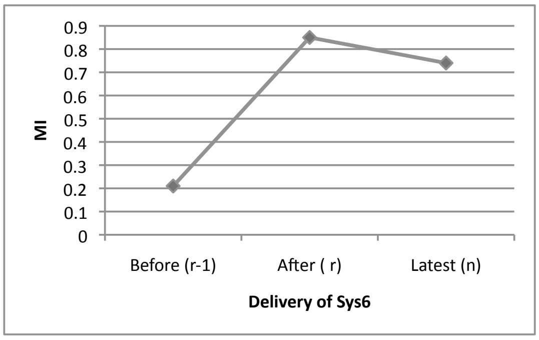
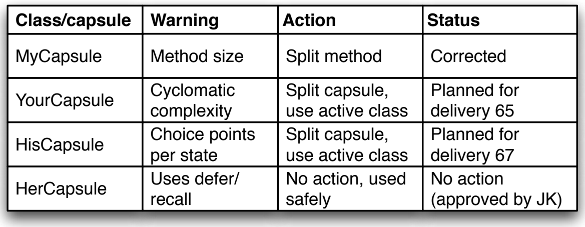


Figure 7: MI for Sys6 Before and After Refactoring

be proposed. This action may be modified by the designer to describe the solution in more detail. Each warning should have a status field, where it can be noted whether it has been fixed or not. Figure 8 is an example of how this could be documented.

As we would rather have the tool find false positives than miss real problems, all the findings that are not easily amended should be manually checked. If a finding is a false positive it should be possible to exclude it from the index. To avoid discarding real findings, a system expert should always approve the discarding of a finding. We consider human intervention to be important in this analysis, as there may be special circumstances that require exceptional coding practices.



Class/capsule	Warning	Action	Status
MyCapsule	Method size	Split method	Corrected
YourCapsule	Cyclomatic complexity	Split capsule, use active class	Planned for delivery 65
HisCapsule	Choice points per state	Split capsule, use active class	Planned for delivery 67
HerCapsule	Uses defer/ recall	No action, used safely	No action (approved by JK)

Figure 8: Maintainability Index Actions

We also strongly suggest that the organization defines a process for maintaining the model, including the definition of the MI function as well as its thresholds. The model should be evaluated regularly and adjusted when necessary. Both the overall maturity of the product and the business goals of the project may be taken into account when defining the model. When a product reaches high maturity, it may be more appropriate with stricter thresholds than when the first versions of the product are created and the business priority may be to hit a market window. The type of product and the estimated time it needs to be under maintenance also affect this decision. A safety critical system with a total life time of several decades needs to display better maintainability than a mobile phone application, which is in use for only a couple of years. The model should be adjusted to reflect these circumstances.

4.1 MI Deployment Level (MDL)

As with any process improvement initiative, the deployment of the new method may take time and need gradual introduction efforts. In order to ensure that the level to which the maintainability index has been analyzed and acted upon is clear to both the team and the project, we suggest a maintainability index deployment level (MDL) to be measured. These are the suggested levels.

Lvl 1: the metrics have been collected.

Lvl 2: the metrics have been collected. Actions for the findings have been suggested.

Lvl 3: the metrics have been collected. Actions for the findings have been suggested and the minor ones implemented.

Lvl 4: the metrics have been collected. Actions for the findings have been implemented or planned for later implementation, including project approval.

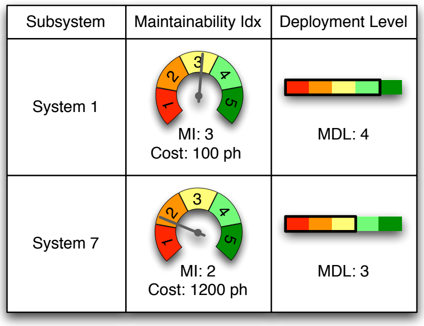
Lvl 5: the metrics have been collected. Actions for all the findings have been taken.

Targets for when the subsystems should have reached the certain levels should be agreed upon.

4.2 Visualization

The MI and MDL can be visualized using a graphical visualization method familiar from, e.g., the balanced score card approach [15]. An example for two subsystems is displayed in Figure 9. For each subsystem, a gauge and a progress bar shows the current status of the subsystem. The cost to improve the MI by one step can be estimated, e.g., by looking at how much the measurements deviate from the threshold and based on historical data of the cost of similar changes. Initially, this number can be based on cost estimates given by the developers.

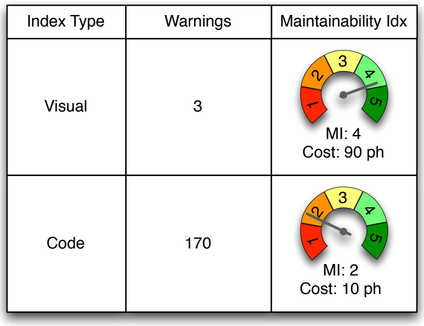
It should be possible to drill down into the subsystem to see exactly where the problem is located. In the example in Figure 10, we have drilled down



Subsystem	Maintainability Idx	Deployment Level
System 1	MI: 3 Cost: 100 ph	MDL: 4
System 7	MI: 2 Cost: 1200 ph	MDL: 3

Figure 9: Maintainability Index Visualization

into the System 1 subsystem and are looking at the visual and code indexes and the number of warnings that are the cause for the index.



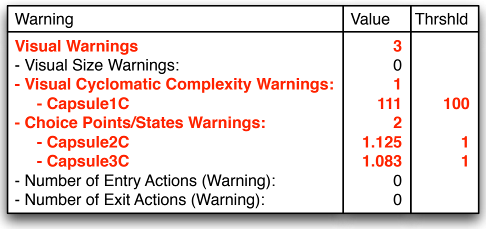
Index Type	Warnings	Maintainability Idx
Visual	3	MI: 4 Cost: 90 ph
Code	170	MI: 2 Cost: 10 ph

Figure 10: Maintainability Index Drilldown

We should be able to further drill down into the two different index types to see the individual capsules and warnings. For an example of this, see Figure 11.

5 Related Work

Different methods for measuring maintainability have been proposed and validated through empirical methods before. Many use an approach similar to ours, including a qualitative element, where subjects evaluate the design or code based on their experience and knowledge of good design [4, 19, 18, 3, 6]. The main difference is the fact that most studies are performed in a small setting with small example software systems (or "toy code application[s]" to quote Mäntylä and Lassenius [18]) and students as subjects, whereas



Warning	Value	Thrshld
Visual Warnings	**3**	
- Visual Size Warnings:	0	
- Visual Cyclomatic Complexity Warnings:	**1**	
- Capsule1C	**111**	**100**
- Choice Points/States Warnings:	**2**	
- Capsule2C	**1.125**	**1**
- Capsule3C	**1.083**	**1**
- Number of Entry Actions (Warning):	0	
- Number of Exit Actions (Warning):	0	

Figure 11: Maintainability Index Drilldown

our study was performed in an industry setting with real systems and experienced software industry professionals as subjects. Although academic studies are invaluable for furthering the state of the art, there are issues that arise specifically when the trying to apply the results in an industrial setting.

Coleman et.al. [6] provide methods for deriving a maintainability index for the purpose of decision support. They also calibrate their models using expert intuition. One difference between their approach and ours is the fact that our focus is more on early diagnostics and actionable metrics, whereas their approach partly relies on after-the-fact metrics such as effort. Although this can be estimated for diagnostic purposes, we chose to focus on a purely internal product metric for our purposes. As an effect our approach provides specific detailed improvement proposal support.

Moha et.al. [20] present a very complete method for specification and detection of smells. They rely on domain experts to identify and specify smells and how they are mesured in the specific context. In academic settings this can be rather easily acieved by, e.g., majority rule. It is our experience, however, that in an industrial setting it may be more difficult for the domain experts to reach consensus by voting, since the most experienced experts are most likely to have some attachment to the system, either by having been involved in the development of the system or at least by knowing they will be involved in it in the future.

Asthana and Olivieri [2] have done similar work in an industry context. Their "Software Readiness Index" has a very different purpose than ours, though. They look at the entire life-cycle of the software system and measure whether it is ready for release. Their focus is more on process metrics, whereas we are more interested in internal product metrics. Furthermore, they use quantitative data only and do not evaluate their results formally. There are many similarities between our approaches, especially due to our wish to provide an organization with decision support.

Our recommendation is to use an approach such as ours to help the organization understand what constitutes maintainability in their domain, capturing the experience of the domain experts and providing the organization with decision support on where to focus their refactoring efforts.

6 Conclusions and Future Work

In this paper we present an approach to assess the maintainability of software systems and their need for refactoring. It is based on the collection of internal product metrics. This work was performed in the context of a large software system in the telecommunications industry and with the help of experts from the industry. The studied subsystems were written using IBM Rational Rose RealTime and C++.

The historic data on one subsystem gave us data points just before and after a major refactoring. It was clear that the maintainability of the subsystem had improved substantially due to this refactoring. This observation was confirmed by our model. Through our model, we could also observe that this subsystem had later seen a slight degradation in maintainability. This confirms the accepted fact that continuous refactoring is needed in order to preserve the maintainability of a software system.

Future work includes long-term evaluation of the impact on software maintainability of methods such as the one defined here, including the evolution of the model itself. The process paradigm in the context of this study was an iterative, incremental one. It would be interesting to see the impact of this method in other contexts. Heidenberg and Porres have presented a method for adopting this approach in an agile and lean context [13] as well.

Acknowledgments

Besides Jeanette Heidenberg and Iván Porres, the project team also consisted of Jussi Auvinen and Andreas Nåls. Marko Toivonen assisted the team with his enthusiastic CLISP coding, which provided us with the tool that we used to collect metrics for our study. We wish to thank the following people in M-MGw management: Stefan Blomqvist, Harri Oikarinen, Seppo Korhonen, Heli Hirvo, Marko Koskinen, Teppo Salminen, Hannu Ylinen.

References

[1] ANSI/IEEE. Std-729-1991: Standard glossary of software engineering terminology. Technical report, IEEE, 1991.

[2] A. Asthana and J. Olivieri. Quantifying software reliability and readiness. In *Communications Quality and Reliability, 2009. CQR 2009. IEEE International Workshop Technical Committee on*, pages 1–6, May 2009.

[3] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[4] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Eng.*, 27(6):513–530, 2001.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[6] Don M. Coleman, Dan Ash, Bruce Lowther, and Paul W. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[7] Ward Cunningham. The wycash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 29–30, New York, NY, USA, 1992. ACM.

[8] Ericsson. Ericsson media gateway for mobile networks (m-mgw). Available at `http://www.ericsson.com/products/hp/Ericsson_Media_Gateway_for_Mobile_Networks__M_MGw__pa.shtml`, 2006.

[9] Norman Fenton and Shari Lawrence Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach.* PWS Publishing Co., Boston, MA, USA, 1997.

[10] F. Fioravanti and P. Nesi. Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems. *IEEE Trans. Softw. Eng.*, 27(12):1062–1084, 2001.

[11] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] Jeanette Heidenberg, Andreas Nåls, and Ivan Porres. Statechart features and pre-release maintenance defects. *J. Vis. Lang. Comput.*, 19(4):456–467, 2008.

[13] Jeanette Heidenberg and Iván Porres. Metrics functions for kanban guards. In *IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2010.

[14] International Standards Organization. Iso 9126: Software engineering – product quality. Technical report, International Standards Organization, 2001.

[15] R. S. Kaplan and D. P. Norton. The balanced scorecard-measures that drive performance. *Harvard Business Review*, pages 71–79, Jan-Feb 1993.

[16] P. Kruchten. *Rational Unified Process*. Addison-Wesley, 1998.

[17] Martin Kunz, Reiner R. Dumke, and Andreas Schmietendorf. How to measure agile software development. pages 95–101, 2008.

[18] Mika V. Mäntylä and Casper Lassenius. Drivers for software refactoring decisions. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 297–306, New York, NY, USA, 2006. ACM.

[19] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Softw. Engg.*, 11(3):395–431, 2006.

[20] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, 2010.

[21] Object Management Group. UML 2.0 Superstructure Specification. Technical report, OMG, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.

[22] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

Turku

Centre *for*

Computer

Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku

- Department of Information Technology
- Department of Mathematics

Åbo Akademi University

- Department of Information Technologies

Turku School of Economics

- Institute of Information Systems Sciences

ISBN 978-952-12-2517-8
ISSN 1239-1891