Seminar Forschungstrends der Softwaretechnik: Maintainability

1st Fabian Egl
dept. name of organization (of Aff.)
name of organization (of Aff.)
Ulm, Germany
fabian.egl@uni-ulm.de

***Abstract*—Software Maintainability is with Functionality, Reliability, Usability, Efficiency and Maintainability one of the main quality characteristics of software engineering based on the ISO-9126. With maintenance being a big part of maintainability, this can be one of the most important qualities of a software system during its lifecycle.**

To make sure a software system has good maintainability, maintainability has to be measured. This paper summarizes multiple metrics and measures for maintainability. To furthermore develop a system with good maintainability in mind as well as maintaining a good maintainability during the lifecycle of a software system approaches for development of software systems are necessary that include maintainability. This paper also gives a short overview of approaches that can be used to guarantee this.

Finally this paper also aims to discuss those metrics and approaches and consider the results of using those for the development process of a software system as well as its stakeholder.

I. Introduction

This section gives a short introduction into the thematic as well as a motivation for this paper. Additionally it explains the definition used for maintainability throughout the whole paper.

Software development is complex an includes many steps. Herby current software systems often have many developers and many changes during the lifecycle of the system. Over time developers can also change. As well as during this time software systems can get many updates. Those updates then can add or remove functionality, fix bugs and even patch security weaknesses. With software systems having long times of use, maintainability of software is an important part of software system starting already during development.

During the further lifecycle of a software system maintainability gets a lot more important, because the development of the basic software system is then finished. And all the further updates and maintenances are based on the previous code. With bad maintainability this results in increased development times and thus increased costs, because developers first have to understand the code before they can change it. To avoid a later escalation in workload and costs when maintaining the system it is thus of utmost importance to start development with maintainability already in mind.

Another Problem is with maintenances in software often new faults are introduced in the system that then need to be fixed in later maintenances.

A. Maintainability

The international standard ISO 9126 is widely used in software development and defines the quality model and the quality characteristics. The six main quality characteristics defined in the ISO 9126 are: Functionality, Reliability, Usability, Efficiency, Portability and Maintainability. This paper focuses on the quality characteristic maintainability.

In ISO 9126 Maintainability is further defined as ability to identify and fix a fault within a software component. In some papers the maintainability characteristic is further referenced as supportability.

In ISO/IEC 25010 software maintainability consists of the quality attributes of: modularity, reusability, analyzeability, modifiability and testability.

B. Motivation

With increasing size of software systems and many developers that can change during the time a software system is in use, bad maintainability can lead to huge problems for later software updates and maintenance. To be able to analyze maintainability of software measures and metrics have to be defined first. That can then be used the measure maintainability for an individual software system. Furthermore general approaches/models for the development process of a software system have to be defined and implemented to guarantee good maintainability already in the initial stages of the development of a software system. As well as during later software updates or software maintenance of the software system.

At least software systems should be developed with maintainability in mind. This avoid problems with maintainability in later phases of the software lifecycle after the development of the software system has finished and the maintenance phase of the software system begins.

C. Goal

The goal of this paper is to give a short overview of different measures and metrics that can be used to measure maintainability. With these maintainability of different systems or maintainability of different versions of the same system can then be compared.

Additionally it aims to give a short overview over different approaches to analyze and improve maintainability in a software system as well as comparing those approaches.

Furthermore it discusses how those approaches could and should be used. And it discusses additionally what results and improvements can be gained by including them in the development process of a software system.

D. Overview

This section contained a short introduction for this paper as well as a general motivation. Section II is a short overview over the methodology that was used in researching for this paper as well as writing it. In section III a few related works are introduced that either have a similar aim as this paper or are about other measures, metrics or approaches for maintainability, which are not covered in this paper. Section IV contains a short overview over basic knowledge and terminology that should be understood before reading this paper.

The main part of the paper contains section V, VI as well as VII. A short overview of measures and metrics that can be used to measure maintainability in a software system is written in section V. This is then followed by different approaches to develop software systems with maintainability in mind in section VI. Section VII discusses the measures and metrics as well as approaches mentioned in section V as well as VI. It furthermore compares them and examines the results in relation to the development process of software systems and the results for stakeholders of software projects.

The final part of the paper section VIII is a short conclusion of the results of this paper.

II. Methodology

This section contains a short overview of the methodology used to research sources to create this paper. Additionally it gives a short overview of the methodology used to write this paper.

First I did research as much papers to the topic maintainability in software systems, I could find. I started with the references from the given papers as well as the papers that did cite the given papers themselves. After that I extracted keywords like maintainability and used those to search for other papers. The first keyword search was on IEEE, the same source the given papers were from. After that I included the results from the keyword search with google scholar as well.

Now with a big base set of possible usable papers I did a first analysis phase. In this phase I removed papers based on their abstracts if they did sound like they were not usable or they just were not about maintainability.

At that point a definition problem did arise, because most of the papers were talking about maintenance and not maintainability. For this I had to define maintainability or use an already existing definition. I then decided to use the definition from the ISO 9126, because that definition is widely used and I did find several mentions of that definition in the abstracts of the papers.

Now with maintainability defined and maintenance included I did filter papers based on maintainability definition again. This resulted in the first usable set of papers.

As next step I did think about structure of my paper. To be able to talk about maintainability of software systems it has to be measured first. Because the ISO 9126 definition does not define any measures or metrics I decided to use this as the first main part of my paper. Based on this I further filtered my set of papers based on measures and metrics for maintainability. This resulted in the first group of papers I used.

Furthermore to guarantee high maintainability while developing a software system as well as having high maintainability during the lifecycle of the software system, approaches to guarantee maintainability are needed. This will be used in the second part of the main part of my paper. Based on this I did filter the papers for approaches that can be used to get high maintainability in a software project. This resulted in the second group of papers I used.

To further reduce those groups I read those papers and tried to understand them and filtered those out that were not useful for the goal of my paper.

The papers from the first group are summarized in Section V. And discussion an comparison of those in Section VII-A.

The papers from the second group are summarized in Section VI. And discussion an comparison of those in Section VII-B.

Lastly I did filter the set of papers again to find papers that have the same goal as mine. Those papers are then mentioned in the related work section as well as papers from the other groups that were not a good fit, but I still wanted to mention them. Those are mentioned in Section III.

III. Related Works

This section is used to give a short overview of other papers that have a similar goal as this paper as well as papers that contain metrics, measures and approaches for maintainability that are not explained in this paper.

The paper by Riaz et al [1] compares and analyzes different software maintainability metrics. This paper has the same goal as this paper in regards to maintainability metrics for software systems.

The paper from Heitlager et al. [2] discusses problems with the maintainability index as. They then creates a new maintainability model.

The following paper mention other approaches to guarantee high maintainability while developing software systems that are not mentioned in my paper.

Velmourougan et al. [3] analyze in their paper the results of application of best practices to provide good maintainability for software systems. Furthermore they define the M-SDLC model that contains best practices and measures that can be taken to minimize errors that can reduce maintainability of the software system.

In their paper Yongchang et. al. [4] analyze and compare different maintenance models in regard to their use for maintainability in a software system.

Durdik et al. [5] create sustainability guidelines to support the development of software systems with maintainability in mind.

Because this paper only gives a general overview of different approaches, for more specialized situations in software development it may be necessary to use different approaches. In case of object oriented systems approaches like van Koten et al. [6] can be used.

IV. Background Knowledge

This section mentions the basic knowledge that should be understood before reading this paper. While it is not necessary to understand everything in this section to also understand this paper, it will help a lot to at least know what the terms are about.

Before reading this paper one should understand basic software metrics like errors/defects/costs per Lines of Code, Lines of Code Mean Time between Failures, Mean time to recover/repair, defect removal efficiency etc.. Furthermore UML and UML Diagrams as basic modeling language should be understood. The main Software development process including the phases and the software lifecycle should be know. This includes functional requirements and other parts of requirements phase.

Basic knowledge of Model Driven Architecture and Model Driven Development should also be understood.

As well as Agile Development including SCRUM.

Part of this paper includes natural language processing, although it does not get explained in detail.

V. Measures and Metrics for Maintainability

This section gives a short overview of different metrics and measures that can be used to measure maintainability and the papers they are from. Maintainability is defined in the ISO 9126 but there are no metrics defined to measure the maintainability of a software system it is necessary to summarize different possible metrics and then analysis them in relation to usability.

A. Metrics for Assessing a Software System's Maintainability

Oman et al. [7] define in their paper metrics that can be used to measure the maintainability of a target software system. The metrics for the target software system are grouped by: software maturity attributes, Source Code and Supporting Documentation.

For software maturity attributes they define:

- Age (Age of Software System since release in months)
- Size (Thousands of Non-Commented Source Statements)
- Stability: $Stability = 1 - ChangeFactor(CF)$ with the change factor being the exponential function of the percent of lines that got changed in each of the last n quarters, with
$$CF = e^D * \sum_{i=1}^{n} c_i/i$$
with $D = C_n - C_1$ and C_i = Percentage of lines changed in each of the last n quarters
- Maintenance Intensity: An exponential function of the Maintenance Activity MA where maintenance activity is the percentage of hours expended in maintenance in each of the last n quarters
$$MI = e^D * \sum_{i=1}^{n} MI_i/i$$
with $D = MA_n - MA_1$, $MA_i = h_i/HPQ_i$, h_i as maintenance hours in each of the last n quarters and HPQ_i as maximum work hours per quarter in each of the last n quarters
- Defect Intensity: An exponential function of the Defect Density DD where defect density is the number of defects reported divided by the Thousands of Non-Commented Source Statements (KNCSS) for each of the last n quarters
$$DI = e^D * \sum_{i=1}^{n} DD_i/i$$
with $D = DD_n - DD_1$ and $DD_i = defects_i/KNCSS_i$
- Reliability: The Musa-Okumoto metric, an exponential function of the Failure Intensity FI where failure intensity is the rate of failures (faults) per hour
$$R(t) = e^{-FI*t}$$
with t as period for which the reliability is estimated
- Reuse (The percentage of lines of code of software system hat have been reused from other systems)
- Subjective Product Appraisal (Evaluations on a 5-point forced-choice scale (very low, low, nominal, high, very high), with the characteristics programming language complexity, application complexity, development effort expended, development techniques used, product requirements volatility, product dependencies, complexity of the software build, completeness of diagnostics test coverage, complexity of the test procedures, installation complexity, intensity of product use and efficiency of the software system)

The metrics for Source code are split into the groups system control structure, information structure and typography, naming and commenting. Following is a short excerpt from the metrics, the complete list can be found in [7] and in figure 1.

1) system control structure
   - Modularity (Number of modules, Average module size)
   - Complexity (V(g), Halstead's E)
   - Consistency (Standard deviation of the module size, Standard deviation of V(g))
   - Nesting (Maximum depth of module nesting, Average depth, Percentage of modules nested)
   - Use of structured constructs (Percentage of single entry single exit structures per module, averaged over all modules)
2) information structure
   - Global data types (Number of global data types divided by the total number of defined data types)

Fig. 1. Target Software System Subtree [7]

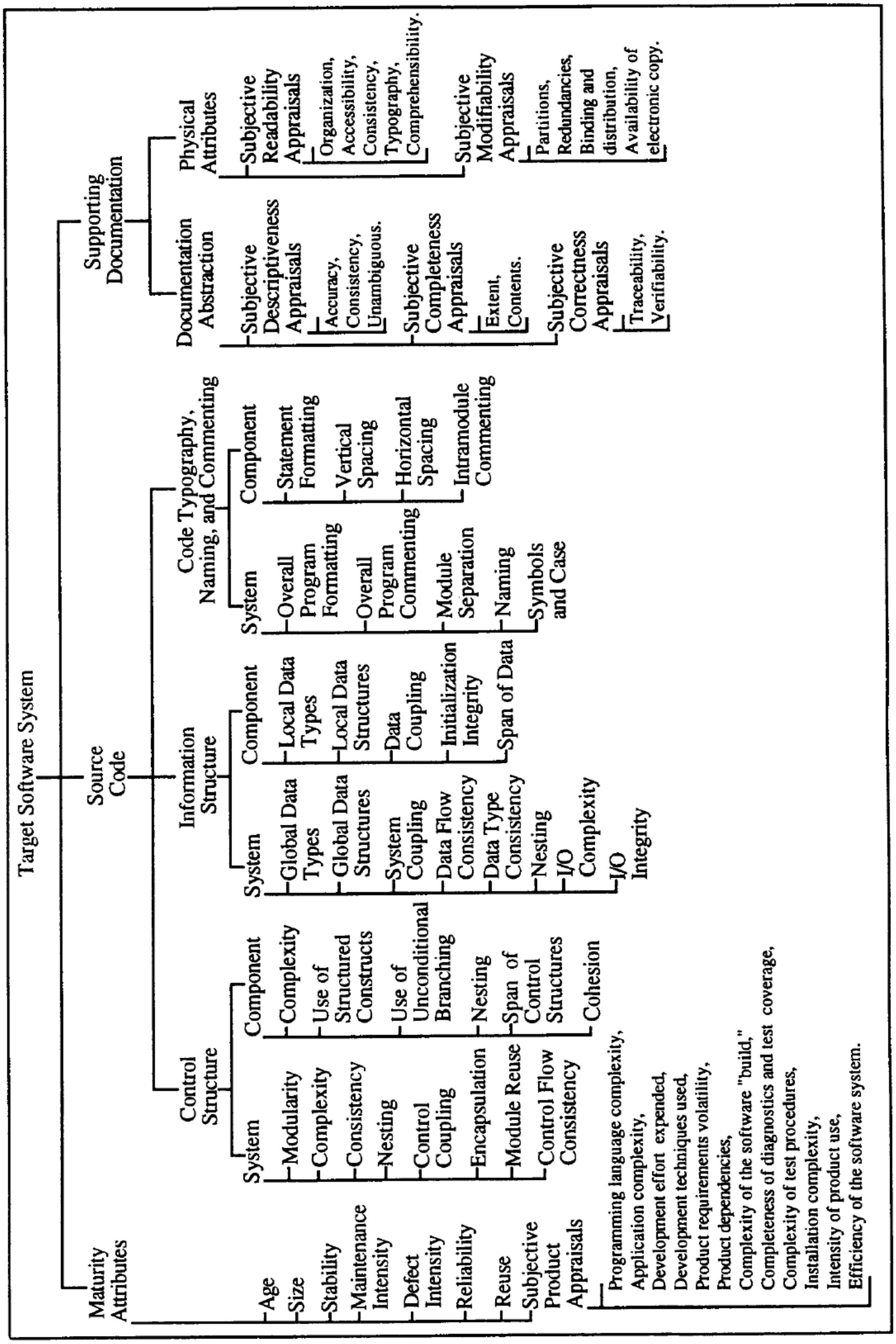

Target Software System
Maturity Attributes
Age
Size
Stability
Maintenance Intensity
Defect Intensity
Reliability
Reuse
Subjective Product Appraisals
Programming language complexity,
Application complexity,
Development effort expended,
Development techniques used,
Product requirements volatility,
Product dependencies,
Complexity of the software "build,"
Completeness of diagnostics and test coverage,
Complexity of test procedures,
Installation complexity,
Intensity of product use,
Efficiency of the software system.
Source Code
Control Structure
System
Modularity
Complexity
Consistency
Nesting
Control Coupling
Encapsulation
Module Reuse
Control Flow Consistency
Component
Complexity
Use of Structured Constructs
Use of Unconditional Branching
Nesting
Span of Control Structures
Cohesion
Information Structure
System
Global Data Types
Global Data Structures
System Coupling
Data Flow Consistency
Data Type Consistency
Nesting
I/O Complexity
I/O Integrity
Component
Local Data Types
Local Data Structures
Data Coupling
Initialization Integrity
Span of Data
Code Typography, Naming, and Commenting
System
Overall Program Formatting
Overall Program Commenting
Module Separation
Naming
Symbols and Case
Component
Statement Formatting
Vertical Spacing
Horizontal Spacing
Intramodule Commenting
Supporting Documentation
Documentation Abstraction
Subjective Descriptiveness Appraisals
Accuracy, Consistency, Unambiguous.
Subjective Completeness Appraisals
Extent, Contents.
Subjective Correctness Appraisals
Traceability, Verifiability.
Physical Attributes
Subjective Readability Appraisals
Organization, Accessibility, Consistency, Typography, Comprehensibility.
Subjective Modifiability Appraisals
Partitions, Redundancies, Binding and distribution, Availability of electronic copy.

Figure 2. Target Software System Subtree

- Global data structures (Number of global data structures divided by the total number of defined data structures)
- System coupling (Number of global structures plus the number of passed parameters divided by the total number of data structures)
- Local data types (Number of local data types divided by the total number of defined data types)
- Local data structures (Number of local data structures divided by the total number of defined data structures)

3) typography, naming and commenting
   - Overall program formatting (percentage of blank lines in the whole program, percentage of modules with blank lines, percentage of modules with embedded spacing)
   - Overall program commenting (percentage of commented lines in the whole program, percentage of modules with header comments)
   - Statement formatting (percentage of uncovered statements per module average over all modules)
   - Vertical Spacing (Number of blank lines divided by the total number of lines in the module, averaged over all modules)

The metrics for supporting documentation are split into the subcategories documentation abstraction and physical attributes as follows:

1) Documentation Abstraction
   - Descriptive Appraisals (Subjective evaluations of the document set for each of the following characteristics: Accuracy, Consistency and Unambiguous)
   - Completeness Appraisals (Subjective evaluations of the document set for each of the following characteristics: Extend of the document set and Contents of the existing documentation)
   - Correctness Appraisals (Subjective evaluations of the document set for each of the following characteristics: Traceability to and from code implementation and Verifiability to actual specifications)
2) Physical Attributes
   - Readability Analysis (Subjective evaluations of the document set for each of the following characteristics: Organization, Accessibility via indices and table contents, Consistency, Typography and Comprehensibility)
   - Modifiability Analysis (Subjective evaluations of the document set for each of the following characteristics: Document set partitions, Document set redundancies, Document set binding and distribution and Availability of electric copy)

They then quantify the the maintainability of the tree as shown in figure 2 with the following formula:

$$\prod_{i=1}^{m} W_{D_i}\left(\frac{\sum_{j=1}^{n} W_{A_j} M_{A_j}}{n}\right)_i$$

with W_{D_i} as Weight of the influence of maintainability Dimension D_i and W_{A_J} as weight of influence of maintainability of Attribute A_j and M_{A_j} as Metric for measure of maintainability of Attribute A_j

B. Using Metrics to evaluate Software Systems

The paper from Coleman et al. [8] shows how to use automated software maintainability analysis. For this they applied two different models for measuring maintainability in software systems to industrial used software. They give a short overview of the models hierarchical multidimensional assessment models, polynomial regression models, an aggregate complexity measure, principal components analysis and factor analysis that can be used to measure maintainability in a software system. They then picked the hierarchical multidimensional assessment model and the polynomial assessment model to apply to real software.

a) A hierachical multidimensional assessment model: This model divides maintainability in three dimensions/attributes: The control structure, the information structure and typography, naming and commenting. They then use metrics for each dimension to measure the maintainability of each category. Those measure can then be combined into an "index of maintainability" for each dimension. Those scores can then be combined again to calculate the maintainability index for the whole software system.

b) Polynomial assessment tools: This model creates a polynomial equation based on the metric attributes for maintainability of a software system. They constructed 50 regression models to find simple generic models that can be applied to most systems. They then found out that the Halstead metrics, Halstead's volume and Halstead's effort metric, were the best metrics for calculating maintainability in a software system.

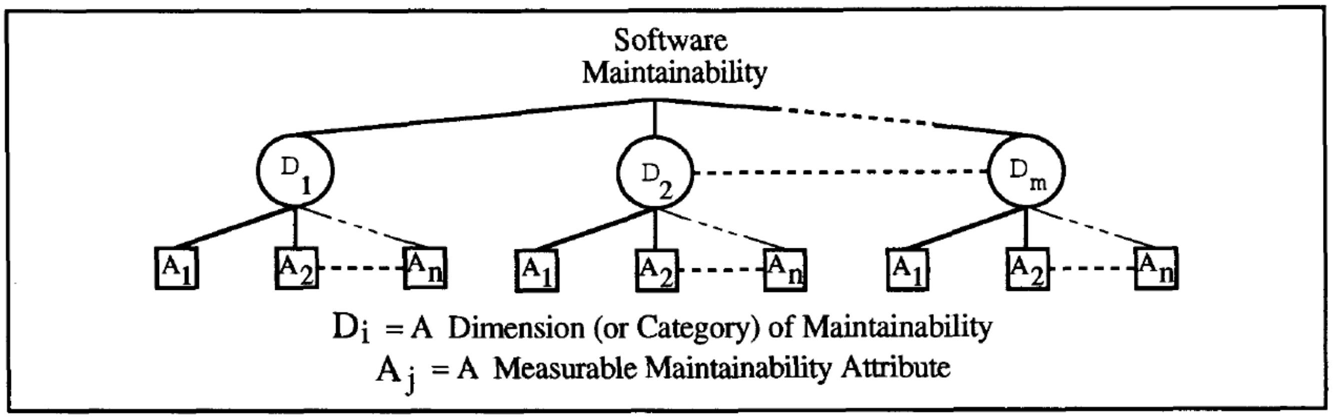
The model that was best applicable to the software system was the following 4-metric polynomial based on Halstead's effort metric:

$$\begin{aligned} Maintainability = 171 \\ - 3.42 * ln(aveE) \\ - 0.23 * aveV(g') \\ - 16.2 * ln(aveLOC) + ave(CM) \end{aligned}$$

with $aveE$ as average effort, $aveV(g')$ as extended $V(G)$, $aveLOC$ average lines of code and $aveCM$ number of comments per submodule.

But this model contains inaccuracies for systems with large amounts of comments. To counteract this, they then defined the following equation for systems with large amounts of comments:

Fig. 2. Software Maintainability Tree [7]


Software Maintainability
D_1
D_2
D_m
A_1
A_2
A_n
D_i = A Dimension (or Category) of Maintainability
A_j = A Measurable Maintainability Attribute

$$\begin{aligned} Maintainability = 171 \\ - 5.2 * ln(aveVol) \\ - 0.23 * aveV(g') \\ - 16.2 * ln(aveLOC) \\ + 50 * \sin(\sqrt{2.46 * perCM}) \end{aligned}$$

with $perCM$ being the percent comments per submodule.

After applying those metrics they found out that applying the metrics resulted in additional information they would not have had without those. But more importantly those result were conform with the opinion of the experts and even supported those.

C. Software Maintainability Index Revisited

Based on Oman et al. [7] [8] (a short overview in the previous sections V-A and V-B) and other papers, K. Welker [9] discusses the uses and usefulness of the maintainability index.

Analyzing the usefulness of comments is subjective and can change over time. Additionally the usefulness of comments can currently only be judged by humans and not with an automated tool. With increasing maintenance, comments that were previously good and informative can become a hindrance in further development, if they were not cared for and updated to include the changed code. This all leads to the point, that just because comments are in the code, this does not mean that the code is maintainable and understandable. Furthermore if comments were helpful in a previous version of the system before maintenances that does not mean that the same comments are still helpful for the current version of the code.

Only measuring the lines of comments in relation to the lines of code poses the additional problem, that documentation outside of comments, for example good helpful names for variables is not included in the metric. But this can be a huge factor in readability and understandability of the code.

Because of this an automated analysis of the source code in relation to maintainability is not possible and subjective opinions of humans, for example developers, are necessary to measure maintainability of software systems.

Another problem with these metrics is the change that has happened in software development over the years. Most of these metrics and measures have been defined before object oriented languages were widely used. Therefore they first have to be adapted to be used in combination with object oriented languages or other new language constructs etc.

All in all the paper concludes that the results of the Maintainability Index has to be interpreted based on the whole system and can't be interpreted in a vacuum alone. It also concludes that code should still be commented. But commented this does not automatically result in an increased understandability and readability of the code and therefore increased maintainability of the software system.

D. An Integrated Measure of Software Maintainability

The paper from Aggarwal et al. [10] proposes an Integrated Measure to measure the maintainability of a software system. This measure includes software readability of the source code, documentation quality and the understandability of the software.

In their approach they define the comments ratio CR as

$$CR = LOC/LOM$$

with LOC being the lines of code including comments and LOC being the lines of comments. This comments ratio can then be used to approximate the readability of the source code.

For the Documentation Quality they did use the Gunning's Fog Index [11]. This index contains the length of sentences and the number of difficult words. The difficulty of a word depends hereby on the number of syllables contained in the word. For good readability the fox index should be low while for bad readability it should be high.

The Understandability of Software is based on a tool proposed by Laitnen [12]. It is based on the similarity between

Fig. 3. context table [13]

Category	Metric	Level
Complexity	Total Lines of Code (TLOC)	P
	Total Number of Files (TNF)	P
	Total Number of Classes (TNC)	P
	Total Number of Methods (TNM)	P
	Total Number of Statements (TNS)	P
	Class Lines of Code (CLOC)	C
	Number of local Methods (NOM) [27]	C
	Number of Instance Methods (NIM) [28]	C
	Number of Instance Variables (NIV) [28]	C
	Weighted Methods per Class (WMC) [29]	C
	Number of Method Parameters (NMP)	M
	McCabe Cyclomatic Complexity (CC) [2]	M
	Number of Possible Paths (NPATH) [28]	M
	Max Nesting Level (MNL) [28]	M
Coupling	Coupling Factor (CF) [30]	P
	Coupling Between Objects (CBO) [29]	C
	Information Flow Based Coupling (ICP) [5]	C
	Message Passing Coupling (MPC) [27]	C
	Response For a Class (RFC) [29]	C
	Number of Method Invocation (NMI)	M
	Number of Input Data (FANIN) [28]	M
	Number of Output Data (FANOUT) [28]	M
Cohesion	Lack of Cohesion in Methods (LCOM) [29]	C
	Tight Class Cohesion (TCC) [31]	C
	Loose Class Cohesion (LCC) [31]	C
	Information Flow Based Cohesion (ICH) [26]	C
Abstraction	Number of Abstract Classes/Interfaces (NACI)	P
	Method Inheritance Factor (MIF) [30]	P
	Number of Immediate Base Classes (IFANIN) [28]	C
	Number of Immediate Subclasses (NOC) [29]	C
	Depth of Inheritance Tree (DIT) [29]	C
Encapsulation	Ratio of Public Attributes (RPA)	C
	Ratio of Public Methods (RPM)	C
	Ratio of Static Attributes (RSA)	C
	Ratio of Static Methods (RSM)	C
Documentation	Comment of Lines per Class (CLC)	C
	Ratio Comments to Codes per Class (RCCC)	C
	Comment of Lines per Method (CLM)	M
	Ratio Comments to Codes per Method (RCCM)	M

the language of the source code and the language of the documents. If they are similar the understandability of the code is high.

The problem with all those factors is, that they are subjective. For this they created a tool that includes all of these factors as well as the subjectivity of those.

For the complete functions used to calculate the Integrated Measure of Software Maintainability see [10].

E. How does Context affect Software Maintainability Metrics

The work from Zhang et al. [13] analyzes and discusses how different context affects different software maintainability metrics. For this they used different software systems from different popular application domains. They then used the context factors of Application Domain, Programming Language, Age, Lifespan, Number of Changes and Number of Downloads for their analysis. To analyze the software systems they did use 39 metrics related to the five quality attributes from software maintainability as defined in ISO/IEC 25010 [13].

After that they did group the metrics in the following categories: complexity, coupling, cohesion, abstraction, encapsulation and documentation. Furthermore these metrics are from the levels: project level (P), class level (C) and method level (M). The full list of metrics used to measure maintainability in their paper can be found in figure 3.

The results of the analysis of the different context factors are that all context factors impact the distribution of the maintainability metrics values. With the programming language being the most important one followed by application domain and lifespan.

They also propose guidelines that can be uses to group software systems into subgroups when analyzing the metrics for the software systems.

VI. Approaches for high Maintainability

In this section approaches and their papers are summarized that can help to develop a software system with maintainability in mind. And to guarantee a good maintainability of the software system during its whole lifecycle.

The first section VI-A is a short overview of analyzing functional requirements with natural language processing to improve the requirements specification in the requirements phase of the software development cycle. In the second section VI-B code gets automatically generated from requirements based on the model driven architecture development approach. The next section VI-C contains a model to include unplanned maintenances into the default SCRUM process to improve the maintainability of the software system. In the final section VI-D the impact of software development problems on maintainability gets discussed.

A. Requires Analysis Based on Software Maintainability

Hu et al. [14] use a classification approach based on natural language processing. They use this to analyze and classify functional requirements statements and cluster the results of those. With help of those they want to reduce the faults in the requirement phase of software development because this phase is the main source of software fault generation. Furthermore most of these faults are from either incomplete software requirements specifications or other problems resulting from the software requirements document.

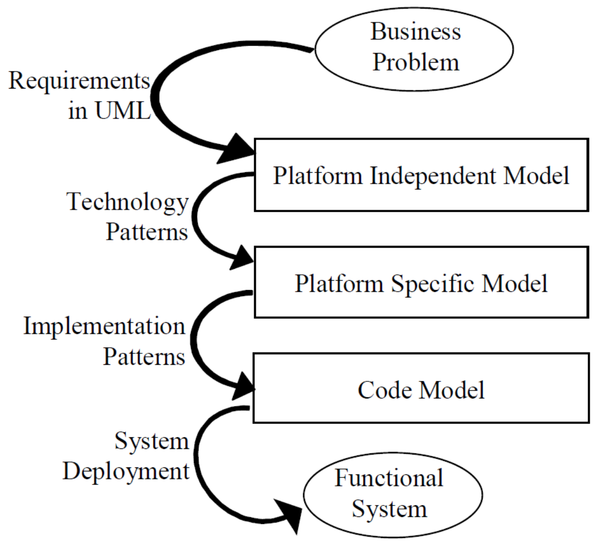
They then start to extract the semantic keywords and remove duplicated and low frequency keywords from the functional requirement statements. After that they generate weight vectors based on the keyword sets that were generated before. Then they use the grey similar correlation and grey closed correlation to construct the closed association matrix between the requirements statements. The results of this analysis can then be used to classify the functional requirement statements from the specification.

Groups of similar functions can then be grouped together. This then results in an improved requirements review. Additionally this is of importance for every other phase of the software development lifecycle because the other phases all build on the requirements formulated in the requirement phase. With the now improved requirement phase the functional requirements can be better implemented including less faults generated by misinterpreted requirements.

B. Code From Requirements

The paper from Bowels et al. [15] uses a Model Driven Architecture approach. With this approach Code can be automatically generated from the model specified with requirements from the requirement phase. Typically the requirements

Fig. 4. Model Driven Architecture development process [15]



are most defined in UML or a modeling language similar to UML. Generating Code from the model can then be used to improve development cycle and reliability.

But this all depends on the rules used to transform the model to code. If the rules are unoptimized, bad or are not conform to basic design patterns it can still result in bad or unoptimized code. This code can then even be worse than code written without the model driven approach.

Code transformation can reduce amount of developers necessary to develop the software system because basic patterns in requirements and code can be repeated through the same transformation rule. With those transformations the resulting code is independent from the different developers with different qualities of code, and depends only on the developers and patterns that are used in generating the transformation rules.

Optimization of the patterns used in the code transformation as well as the requirements specification can lead to improved code for the whole project if the patterns can be used multiple times. Furthermore if one pattern changes, the transformation rules for that pattern can be adjusted, and the then generated code will contain the updated pattern throughout the whole code. This means there is no need to search the whole code for the changed pattern and update it there. The code only has to recreated once with the updated transformation rules.

This furthermore reduces the mistakes that could happen if not all patterns that have to be changed are found in the code. And there can not be any inconsistencies that can happen if different developers try to fix the same faulty code in different parts of the system.

The model driven architecture development processes used in the paper is based on figure 4.

Another advantage of using the Model Driven Architecture approach is the independence from the target systems or other requirements of the system like programming language etc.. All of this results in more resources available to focus on the problem that should be solved with the software system and not the development of the software system itself. With improved focus on the requirement phase in the development cycle the number of faults of a software system can be greatly reduced, because most faults happen in the requirements phase.

To make sure that the resulting code is well formed and understandable code, the transformation rules to code have to be written by specialized and skilled people and should be used on widely used and good programming patterns. With the use of those people for the code generation and the focus on good code generation, the code will be of a high standard, good written and be consistent throughout the whole project. Which results in a high maintainability of the software system during its whole lifecycle.

C. SCRUM Software Maintenance Model

The paper from Rehman et al. [16] considers SCRUM and Agile Methods in relation to maintainability of a software project.

They first added a separate request for corrective maintenance task that should be started immediately to SCRUM. The developers then started with an agile version of maintenance, but quickly noticed that this resulted in issues. This happened because maintenance and development are different process that can not be substituted for each other.

For this they proposed a SCRUM Software maintenance model as seen in figure 5.

Additionally to the normal sprint this model contains the possibility for short and unplanned sprints. These sprints can be started any time in case of emergency maintenance work that has to be done. To be able to plan those unplanned sprints they added a small buffer of time during the normal sprint that is not allocated to any task.

In this model if need for an emergency maintenance arises the normal sprint gets paused and stored in version control. Then the emergency sprint gets started on a new version of the system. After the emergency sprint has finished the paused sprint gets resumed again.

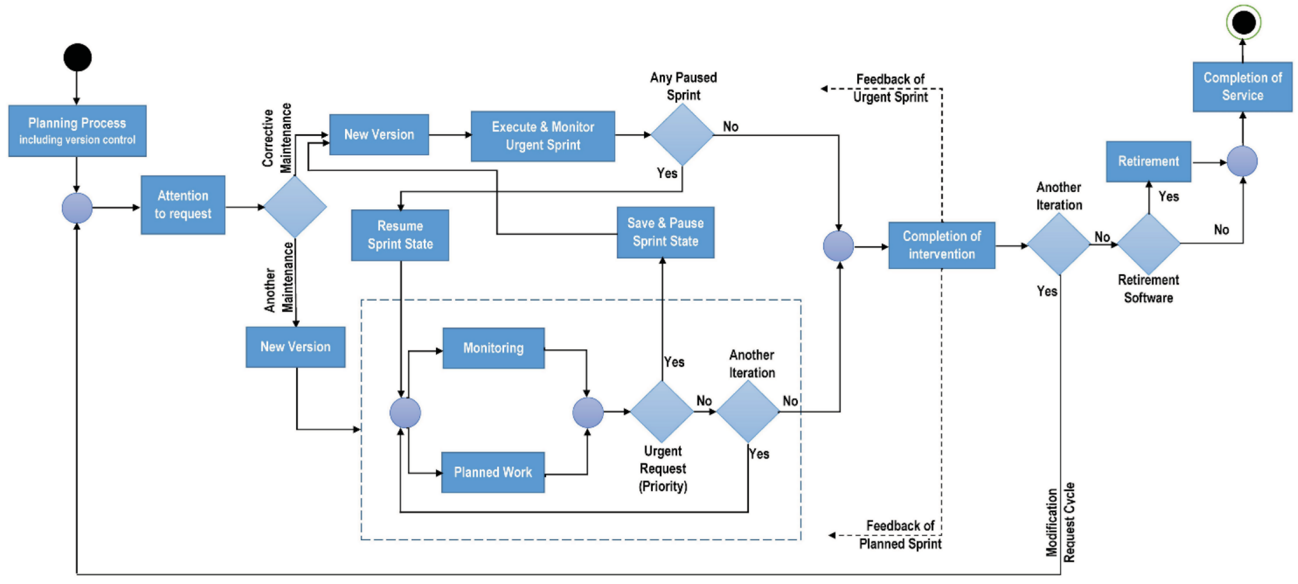
Important points of this model include the planning for maintenance, a documentation update after each phase and the testability of the maintenance.

The application of this model resulted in increased client as well as maintenance engineers satisfaction. The separate version for each sprint also made testing easier.

D. Impact of Software Development Problems on Maintainability

The paper from Chen et al. [17] classifies and analyzes 25 different problem factors in regard to their influence on software maintainability. Those factors were grouped into the groups: documentation quality problems, programming quality problems, system requirements problems, personnel resources problems and process management problems.

Fig. 5. SCRUM Software Maintenance Model [16]



To analyze those factors, they applied them to real systems. The main findings of the analysis were. The severity of software development problems have to be decreased to improve the maintainability of the software project. Dealing with the documentation quality problems and programming quality problems is most important to increase the maintainability of the software system. Dealing with personnel resources problem only really helps in systems with really bad maintainability. The better the maintainability of a software system gets, the less it improves when dealing with personnel resources problems.

VII. Discussion

This section contains a short overview of the results of the different measures and metrics as well as the different approaches that are summarized in this paper. In section VII-A measures and metrics from this paper are discussed, with focus on how to improve maintainability in a software system. In section VII-B the different approaches summarized in this paper are discussed in regard to use-cases and when to use them. As well as their results on the development process or the software system itself. In sections VII-C and VII-D the results for the development process and stakeholders get discussed.

A. *Measures and Metrics*

In this paper many metrics and measures for maintainability in software systems have been summarized. The basic metrics like Lines of Comments in Code, they are subjective. Especially comments that are often used in relation to maintainability and readability/understandability of code can not be used as a good measure.

Comments in code are completely reliant on the information they posses in regard to the code part they belong to and can depend on changes in systems. For example if code gets changed during maintenance, but the comment for the code stays the same, there might pe inconsistencies between code and comment. They then can reduce the understandability of the code and might actually increase the time for the next maintenance, which leads to worse maintainability.

For this it is necessary to keep the comments of the code up to date in relation to their code part.

Furthermore if code comments are kept close to the documentation in language and functionality, they help maintainability. But it should be avoided to have big differences in language or functionality of comments and documentation, because this leads to worse understandability.

Factors as readable variable names, that tell the reader instantly what they do can increase the understandability of code and thus result in better maintainability. This can not be analyzed automatically by current tools. To include these factors, developers or expert have to asked to examine the code.

Because the requirements phase is most error prone the focus to reduce faults in the system should primary be in the requirements phase. This reduction of faults in the requirement phase, then leads to increased maintainability, because then they don’t have to be fixed in later maintenances.

In their work Zhang et al. [13] analyzed different context factors and their influence on the maintainability on software systems. They found that the programming language is the most important context factor in regard to maintainability followed by application domain and lifespan of the system. Furthermore the other factors they analyzed, age of the system, number of changes as well as number of downloads, all had a big relation to maintainability of the software as well.

B. Approaches

Possible approaches to increase maintainability in a software project that are summarized in this paper are natural language processing for the functional requirements, a model driven architecture approach to generate code from requirements and a SCRUM model with added support for maintenance sprints.

Hu et al. [14] used natural language processing to analyze and classify functional requirements. This then results in improved requirements which furthermore improves the other phases of the software development cycle as well, because they all are build on the requirement phase.

Bowels et al. [15] used model driven architecture to transform code from requirements specification (for example UML) to code. This transformations are based on general patterns and should only be done by skilled programmers or expert, that can maintain high code quality. Because those transformation patterns can be applied multiple times if there are similar functions in the requirements.

This approach results in more uniform code, because each of the used programming pattern is written only once in the transformation rule, and then applied each time the pattern is needed in the finished code. In a software project without the model driven approach there are potentially multiple developers, that all code the same patterns but in different qualities.

It also results in improved maintainability, because if the pattern has to be changed in a maintenance during the later lifecycle of a software system, the newly generated code contains the changed pattern every where. This means code only needs to be changed once in a transformation rule.

Rehman et al. [16] propose a new model for SCRUM that includes maintenance sprints. Those sprints can additionally be emergency sprints for critical maintenance, that can pause the normal sprint. To be able to run those emergency sprints short time blocks of a sprint are kept without task.

This model has the advantage, that maintenance is planned into the sprint. Furthermore even the unplanned critical maintenance is planned into the model. The handling of different versions for normal sprint and emergency maintenance sprint, increases the testability of the code.

C. Results for the Development Process

Most faults of a software system come from the requirement phase. Those faults then have to be fixed in later maintenance and thus result in bad maintainability of the software system. With bad maintainability the necessity for later maintenances to fix problems in the code or bad code increases. With increasing maintenance the development time and the costs of the software project increase as well.

Thats why its important to start the development process with maintainability already in mind and set a huge focus on the requirements phase and the requirements specification, to reduce the faults happening in that phase and that way in the later phases.

When developing agile with SCRUM using the SCRUM Software Maintenance model from Rehman et al. [16] 4 can improve maintainability of the software system by including the maintenance in the development process and the sprint planning.

An abstraction through models can help as well. With the model driven approach the code can be automatically generated by the defined transformation rules that transform the requirements specification into code. In this approach it is of utmost importance, that the transformation rules are written by a highly skilled developer to avoid mistakes. This is important, because those rules contain patterns from the requirements that are matched to code patterns in the generated code. Meaning it can be used a lot of times. If the transformation rules aren't written well this can lead to bad code at lots of parts of the system.

The model driven architecture approach can result in code that is more readable by a huge margin, because the patterns are coded uniform through the whole project, because they stem from the same transformation rule. Furthermore if they are written by a highly skilled programmer the quality of the code is higher than that of a less skilled programmer in a normal software project.

If patterns or requirements change later, then a maintenance can potentially be more efficient. Because if the transformation rule gets changed, each occurrence of generated code from that rule will get change with the next code generation as well. Without model driven architecture each of those code parts have to be changed manually, which can easily lead to inconsistencies if multiple programmers have to do it. Or even parts of code that should be changed, that get missed and remain the same. They then have to be fixed in a later maintenance, reducing the maintainability of the code.

To summarize, with increased maintainability of the software system, the necessity for later maintenances is reduced. Leading to less developer time to fix those problems and thus reduced cost for the maintenance of the software system during its lifecycle.

D. Results for Stakeholder

Increasing the focus on the requirement phase to reduce faults, leads to less maintenance in the later lifecycle of the software system. This means developers need to invest less time to fix those faults, and this results in reduced costs for the company as well as the clients.

The planned maintenance phases in the SCRUM Software Maintenance Model also reduce the time developers spend on unplanned maintenance phases and the organization of those. This results in better planning, time management for the developers as well as the maintenance engineers. With clients in loop for the sprint meeting there exists reduced complexity in the communication of the unplanned maintenances. As well as less faults by misunderstood requirements.

Model driven architecture can have advantages for the company and developers as well, because they are not limited based on software requirements like operating system. They are only necessary for the transformation rules but not for the model of the architecture itself. That means they can build the model for the system with a lot more freedom. The requirements like operating system or programming language are only important for the transformation itself.

This also introduced the possibility to develop one system for multiple environments. Meaning only the few transformation rules that are specific to the requirements have to be change, all the other transformation rules stay the same. This can lead to further reductions in development cost for the company.

VIII. Conclusion

This section contains a short conclusion of the results of this paper. As well as future implementations or improvements.

To develop a software system with good maintainability it is important to reduce the faults in the requirement phase already. This is necessary because most faults happen in the requirements phase. And all those faults have to be fixed in later maintenances during the software lifecycle costing time and money.

This paper contains a short overview of measures and metrics that can be used to measure the maintainability of a software system. But most of those metrics are subjective. This means that depending on the software system different metrics should be used. Furthermore the metrics can not be analyzed in a vacuum, they need context in relation to the software system as well as human experts. But still there are automatic metrics that can be used to support the measuring of the maintainability of a software system.

Additionally different approaches can be used to develop a software system with maintainability in mind from the beginning.

Natural language processing can be used to improve the functional requirements of the requirements specification.

The SCRUM Software Maintenance Model can be used to include maintenances in the sprint planning.

Model Driven Architecture can be used to generate code from models with abstraction in relation to software requirements. Code can then be generated from the model with transformation rules written by highly skilled programmers.

This paper only contains a small overview of metrics and measures as well as approaches for maintainability in a software system. With many of those based on old systems, with technologies like object oriented programming not included. In a future work those could be included.

References

[1] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE, 2009.

[2] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007.

[3] S. Velmourougan, P. Dhavachelvan, R. Baskaran, and B. Ravikumar. Software development life cycle model to improve maintainability of software applications. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 270–273, 2014.

[4] R. Yongchang, X. Tao, L. Zhongjing, and C. Xiaoji. Software maintenance process model and contrastive analysis. In *2011 International Conference on Information Management, Innovation Management and Industrial Engineering*, volume 3, pages 169–172, 2011.

[5] Z. Durdik, B. Klatt, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss. Sustainability guidelines for long-living software systems. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 517–526, 2012.

[6] C. van Koten and A.R. Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 48(1):59 – 67, 2006.

[7] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society, 1992.

[8] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[9] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001.

[10] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, pages 235–241, 2002.

[11] James F Peters and Witold Pedrycz. *Software engineering: an engineering approach*. John Wiley & Sons, Inc., 1998.

[12] Kari Laitinen. Estimating understandability of software documents. *ACM SIGSOFT Software Engineering Notes*, 21(4):81–92, 1996.

[13] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How does context affect the distribution of software maintainability metrics? In *2013 IEEE International Conference on Software Maintenance*, pages 350–359, 2013.

[14] W. Hu, S. Wu, M. Zhao, and J. Yang. Requires analysis based on software maintainability. In *2014 10th International Conference on Reliability, Maintainability and Safety (ICRMS)*, pages 354–357, 2014.

[15] J. B. Bowles. Code from requirements: new productivity tools improve the reliability and maintainability of software systems. In *Annual Symposium Reliability and Maintainability, 2004 - RAMS*, pages 68–72, 2004.

[16] Fateh ur Rehman, Bilal Maqbool, Muhammad Qasim Riaz, Usman Qamar, and Muhammad Abbas. Scrum software maintenance model: Efficient software maintenance in agile methodology. In *2018 21st Saudi Computer Society National Computer Conference (NCC)*, pages 1–5. IEEE, 2018.

[17] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.