Automated Quality Defect Detection in Software Development Documents

Andreas Dautovic, Reinhold Plösch
Institute for Business Informatics - Software Engineering
Johannes Kepler University Linz
Altenberger Straße 69, 4040 Linz, Austria
andreas.dautovic | reinhold.ploesch@jku.at

Matthias Saft
Corporate Technology
Siemens AG
Otto-Hahn-Ring 6, 81739 Munich, Germany
matthias.saft@siemens.com

Abstract—**Quality of software products typically has to be assured throughout the entire software development life-cycle. However, software development documents (e.g. requirements specifications, design documents, test plans) are often not as rigorously reviewed as source code, although their quality has a major impact on the quality of the evolving software product. Due to the narrative nature of these documents, more formal approaches beyond software inspections are difficult to establish. This paper presents a tool-based approach that supports the software inspection process in order to determine defects of generally accepted documentation best practices in software development documents. By means of an empirical study we show, how this tool-based approach helps accelerating inspection tasks and facilitates gathering information on the quality of the inspected documents.**

Keywords-quality defect detection; software development document; tool-based approach; software inspection

I. INTRODUCTION

Software quality assurance aims at ensuring explicitly or implicitly defined quality goals for a software product. Assuring the quality of a software product basically deals with the fulfillment of specified functional and quality requirements, where the checks are often realized by static and dynamic testing of the software product. Software development documents like requirements specifications, which define how to build the right software product, or design documents, which define how to build the software product right, are also an essential part of the entire software product. However, they are often not treated with the same enthusiasm as source code. Consequently, software bugs are fixed in a later phase of the software product life-cycle, which leads to increased costs for software changes [1]. For instance, a cost/benefit-model reveals that due to the introduction of design inspection 44 percent of defect costs compared to testing alone can be saved [2]. Therefore, to positively influence the development of a software product, quality assurance also has to systematically deal with the quality of software development documents.

Natural language is a commonly used representation for software development documents. However, as a result of its informal nature, natural language text can easily lead to inadequate or poor project documentation, which makes software hard to understand, change or modify. Based on a comprehensive literature study, Chen and Huang [3] identified five quality problems of software documentation:

- Documentation is obscure or untrustworthy.
- System documentation is inadequate, incomplete or does not exist.
- Documentation lacks traceability, as it is difficult to trace back to design specifications and user requirements.
- Changes are not adequately documented.
- Documentation lacks integrity and consistency.

In order to improve the overall quality of natural language project documents throughout the software life-cycle, the use of inspections is generally accepted. Since the introduction of inspections in the mid-1970s by Fagan [4], some modifications have been made to the original process. Improved reading techniques including: checklist-based reading [5] [6], usage-based reading [6] [7] or perspective-based reading [8] [9] are nowadays available for checking consistency and completeness of natural language texts. However, analysis [10] [11] show that currently available inspection methods are mainly used for source code reviews. This is surprising and can be explained by the lack of tools that fully support software inspections [12] [13], especially in dealing with specific artifact types and locating potential defects in this artifacts. Furthermore, high inspection costs due to the resource-intensive nature of reviews and tedious searching, sorting or checking tasks often restrain the application of software inspections [11].

In this paper we present a tool-based approach that tries to identify potential document quality defects. This tool-based analysis relies on best practices for software documentation. In section II we give an overview of related work in the context of software inspection and document quality defect management tools. Section III shows how documentation best practices can be used to identify document quality defects. In section IV we present our developed tool-based document quality defect detection approach. Section V gives an overview of the results of an empirical study where we used our approach to detect document quality defects in real-world project documentation. Finally, in section VI we give a conclusion and discuss further work.

II. RELATED WORK

In this section we give an overview of existing tools that can be used in the document inspection process. As there are different criteria for categorizing and distinguishing inspection tools [12] [13], we focus on tools that directly address data defects, i.e. tools that enable locating potential quality defects in the documents. Following, we also discuss work of software engineering domains apart from software inspections, but enable comprehensive document quality analysis and assessments. However, tools that support e.g. the collaborative inspection process or the process improvement are out of scope of this work and will not be discussed in this section.

Wilson, Rosenberg and Hyatt [14] present an approach for the quality evaluation of natural language software requirements specifications, introducing a quality model containing eleven quality attributes and nine quality indicators. Furthermore, a tool called ARM (Automatic Requirements Measurement) is described, which enables performing analysis of natural language requirements against the quality model with the help of quality metrics. Lami and Ferguson [15] describe a methodology for the analysis of natural language requirements based on a quality model that addresses the expressiveness, consistency and completeness of requirements. Moreover, to provide support for the methodology on the linguistic level of requirements specifications, they present the tool QuARS (Quality Analyzer of Requirements Specifications) [16]. Further tools that also support the automatic analysis of natural language requirements documents are described e.g. by Jain, Verma, Kass and Vasquez [17] and Raven [18]. However, all these tools are limited to the analysis of requirements specifications that are available as plain text documents. Although, the quality of a software project strongly depends on its requirements, there are a number of additional document types and formats that have to be considered throughout the software development life-cycle.

Farkas, Klein and Röbig [19] describe an automated review approach for ensuring standard compliance of multiple software artifacts (e.g. requirements specifications, UML models, SysML models) for embedded software using a guideline checker called Assessment Studio. The tool performs checks on XML-based software artifacts by using rules formalized in LINQ (Language Integrated Query) [20]. As they use XML as a common file-basis, their approach is not limited to one specific document type or format. Moreover, traceability checks of multiple software artifacts are facilitated. Nödler, Neukirchen and Grabowski [21] describe a comparable XQuery-based Analysis Framework (XAF) for assuring the quality of various software artifacts. XAF enables the specification of XQuery analysis rules, based on standardized queries and pattern matching expressions. In contrast to the approach presented in [19], XAF uses a facade layer for transforming XQuery rules to the individual XML representation of the underlying software artifact. As a result of this layered architecture, XAF enables the creation of re-usable analysis rules that are independent from the specific target software artifact.

III. DOCUMENTATION BEST PRACTICES FOR MEASURING DOCUMENT QUALITY

International documentation and requirements specification standards like NASA-STD-2100-91 [22], IEEE Std 830-1998 [23], IEEE Std 1063-2001 [24], ISO/IEC 18019:2004 [25], and ISO/IEC 26514:2008 [26] provide best practices and guidelines for information required in software documentation. Most of these documentation standards focus on guidelines for technical writers and editors producing manuals targeted towards end users. Hargis et al. [27] focus on quality characteristics and distinguish nine quality characteristics of technical information, namely “task orientation”, “accuracy”, “completeness”, “clarity”, “concreteness”, “style”, “organization”, “retrieveability”, and “visual effectiveness”. Moreover, they provide checklists and a procedure for reviewing and evaluating technical documentation according to these quality characteristics. In order to determine the quality of project documents, Arthur and Stevens [28] identified in their work four characteristics (“accuracy”, “completeness”, “usability”, and “expandability”) that are directly related to the quality of adequate documentation. Nevertheless, documentation quality is difficult to measure. Therefore, Arthur and Stevens [28] refined each documentation quality attribute to more tangible documentation factors, which can be measured by concrete quantifiers. In order words, similar to static code analysis, some quality aspects of project documentation can be determined by means of metrics. Moreover, we think that violations of documentation best practices or generally accepted documentation guidelines can also serve as measurable quantifiers. Consequently, violations of defined rules, which represent such best practices or guidelines, can be used for determining quality defects of documentation.

So far we have identified and specified more than 60 quantifiable documentation rules. Most of these document quality rules cover generally accepted best practices according to the documentation and requirements standards mentioned above. In order to get a better understanding about document quality rules, we show four typical examples. Furthermore we try to emphasize the importance of these rules for software projects, as well as the challenges of checking them automatically.

A. Adhere to document naming conventions

Each document name has to comply with naming conventions. The usage of document naming conventions helps recognizing the intended and expected content of a document from its name, e.g., requirements document, design specification, test plan. A consistent naming scheme for documents is especially important in large-scale software projects. However, defining generally accepted naming conventions for arbitrary projects is not always simple and requires support for easy configuration of a specific project.

B. Each document must have an author

Each document must explicitly list its authors, as content of documents without explicit specified authors cannot be traced back to its creators. This is important e.g., for requirements documents in order to clarify ambiguous specifications with the authors. However, identifying document content particles describing author names is difficult and needs sophisticated heuristics.

C. Ensure that each figure is referenced in the text

If a figure is not referenced in the text, this reference might either be missing or the intended reference might be wrong. Typically, many figures are not self-explanatory and have to be described in the text. It is good style (e.g., in a software design document), to explain a UML sequence diagram or a class diagram. In order to make this explanation readable and consistent, it must always be clear which specific UML artifacts are explained in the text.

D. Avoid duplicates in documents

Within one document duplicated paragraphs exceeding a defined length should be avoided and explicitly be referenced instead, as duplicates make it difficult to maintain the document content. Therefore, word sequences of a specified length that are similar (e.g., defined by a percent value) to other word sequences in the same document violate this rule. However, the defined length of the word sequence strongly depends on the document type and differs from project to project.

IV. THE AUTOMATED DOCUMENT QUALITY DEFECT DETECTION APPROACH

As shown in section III, the identification of documentation defects in software development documents can rely on finding violations of document quality rules, which represent generally accepted documentation best practices and guidelines. However, manual checks of these rules can be very resource and time consuming, especial in large-scale software projects. Due to this, we developed a document quality defect detection tool, which checks software development documents against implemented document quality rules.

Similar to existing static code analysis suites for source code, our tool analyzes document information elements to find out, whether documents adhere to explicitly defined documentation best practices. In contrast to approaches and tools mentioned in section II, our document quality defect detection tool is not restricted to elementary lexical or linguistic document content analysis. Furthermore, it is also not limited to specific software development artifacts but covers the range from requirements across system, architecture and design, up to test specifications. The introduction of open, standardized document formats like Office Open XML [29] or

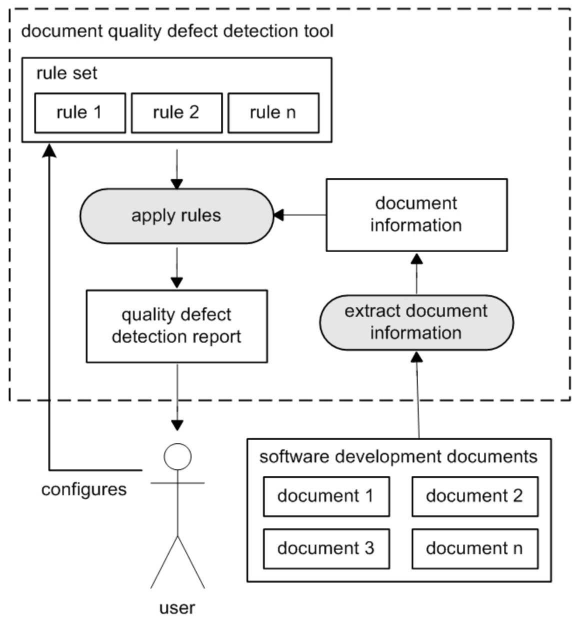
document quality defect detection tool
rule set
rule 1
rule 2
rule n
apply rules
document information
quality defect detection report
extract document information
configures
user
software development documents
document 1
document 2
document 3
document n

Figure 1. Conceptual overview of the document quality defect detection tool usage process

Open Document Format [30] has enabled the extraction of document information in a way, which is beyond unstructured text. In fact, our tool facilitates beside content quality analysis also the check of document metadata like directory information and version information. The use of standardized and structured document models also allows checking more specific content attributes based on the meaning of specific document particles. Moreover, it enables traceability checks to prove document information for project wide consistency. In order to support inspectors in their task, the tool can therefore be used to automatically check cross-references within the document under inspection as well as from the document under inspection to other related documents. The latter aspect is especially important for identifying missing or broken relations between e.g., design documents and software requirements specifications.

Fig. 1 gives a conceptual overview of our developed tool and describes the process of quality defect detection. First of all, the tool user (e.g. project manager, software inspector, quality manager) has to choose the software development documents as well as the document quality rules respectively rule sets, which will be used for the defect detection. If necessary, the selected rules can be configured by the user to meet defined and project specific documentation requirements. After the user has started the tool, all relevant information is extracted from the selected software development documents. The information is represented as a hierarchical data structure containing information of specific document elements (e.g. sections, paragraphs, references, figures, sentences). In a next step, each rule will be applied onto this document information to check whether the document adheres to the rule conditions. Finally, all detected potential document quality defects are linked to the original document to provide a comprehensive quality defect detection report to the user.

As software development documents can exist in many different document formats, considering each format for automated quality defect detection with our tool is challenging. Due to this we developed in a first step a tool that is applicable for Open Office XML documents [29] and Microsoft Office Binary Format documents [31], as these are one of the most commonly used formats for software development documents. Furthermore, these standardized document formats provide access to particular document information that is required, as some rules are applied on specific document elements. Consequently, a traversal strategy to visit all these elements is needed. Due to this, we have implemented the visitor pattern [32] [33] for our tool. Using this pattern, which provides a methodology to visit all nodes of a hierarchical data structure, enables applying rules on each specified element of the extracted document information. A similar rule-checking mechanism is used by the Java source code measurement tool PMD [34]. However, instead of using software development rules for source code, our document quality defect detection tool uses this methodology in order to check software development documents by means of easily adaptable and highly configurable document quality rules.

V. Feasibility Study of the Automated Document Quality Defect Detection Approach

This section describes the results of a feasibility study conducted to test, whether an automated quality defect detection tool using 24 document quality rules is able to reveal additional documentation defects human inspectors did not find before. Furthermore the trustworthiness of these quality rules is shown as well as the effort to fix documentation defects and rule settings. Before, we will give in (A) a brief description of the software project documentation we used in our study and in (B) an overview of the applied document quality rules.

A. Description of the used software project documentation

In order to get appropriate software development documents for our feasibility study, we used project documents of a real-world software project. The software as well as the associated documentation was developed by Siemens AG Corporate Technology in several iterations using a semi-formal development process. The software is used for monitoring the communication and control flow in distributed applications. As we have our focus on the quality of software development documents, more technical or organizational background information of the project is not necessary for the purpose of our study.

TABLE I. SOFTWARE PROJECT DOCUMENT FORMAT TYPES

document format type	no. documents in project
DOC	124
XLS	11
PPT	37
PDF	7

The entire documentation of the software project contains of 179 software development documents of four different document format types. As it is shown in Table I, more than two-third of them are Microsoft Office Word Binary Format documents. However, some of them are internal documents with intentionally lower quality. Due to this, we used a set of 50 officially published Microsoft Word project documents consisting of different document types (requirements specifications, systems specifications, concept analysis, market analysis, delta specifications, function lists, user documentation, etc.) as objects of analysis for our feasibility study. These 50 documents should meet high documentation quality standards and are already checked by software inspectors. Therefore, they testify to be of a high maturity level and ready to be checked by our tool.

B. Applied document quality rules

Following, we list and give a short description for all document quality rules we used in our study and motivate their importance for software development documents. However, the used rules settings are not discussed in this work.

- *ADNC - Adhere to Document Naming Conventions*: Each software development document name has to comply with explicitly specified naming conventions, as project members can better grasp the document content if documents follow a defined project-wide document naming scheme.
- *ADNS - Avoid Deeply Nested Sections:* Documents should not contain a deeply nested section hierarchy. Particularly in software development documents the content structure should be flat, simple and clear in order to support clarity.
- *ADUP - Avoid Duplicates in Document:* Similar to duplicated source code, within software development documents duplicated paragraphs exceeding a defined length (number of characters) should be omitted and are better referenced explicitly. Otherwise the document content will be more difficult to maintain.
- *AES - Avoid Empty Sections:* Each section of a software development document must contain at least one sentence, otherwise the content may not be complete or lacks of clarity and conciseness.
- *AESD - Avoid Extremely Small Documents:* Extremely small software development documents are indicators for unfinished content or for a bad project-wide document structure, as small software development documents might be better combined to larger document of reasonable size.
- *AIDOC - Avoid Incomplete Documents:* Particularly in later phases of the software development process documents should contain all information that is required. Therefore, documents that are formally incomplete, i.e., contain phrases like “TBD” or “TODO” are not yet complete by definition.

- *ALS - Avoid Long Sentences:* Identify those sentences in a project document that exceed a given length, where length is expressed by the number of words contained in the sentence. Long sentences harm the readability of e.g. requirements specifications or test plans and are therefore indicators for difficult to understand content of software development documents.
- *AULD - Avoid Ultra Large Documents:* Ultra-large software development documents should be avoided as they are more difficult to maintain and to keep consistent. Furthermore, it is harder to check whether all information needed is present.
- *ARHT - Avoid Repeated Heading Text:* In a software development document paragraphs of a section should not only consist of a copy of the heading text, as this a indicator of a underspecified and incomplete section
- *ASPE - Avoid Spelling Errors:* Each software development document should be free of spelling errors, regardless whether it is written in one language or contains a mix of languages.
- *ATSS - Adhere To Storage Structure:* Each software development document should be put in the right place of the storage system, i.e. it should typically be stored in a directory according to project-wide rules (typically for different types and/or phases of the software development process).
- *DESOR - Define Expected Skills Of Readers:* For each software development document the skills of readers should be explicitly defined, as depending on the skills of readers, the content of the software development document has to be presented in a different way. So, depending on the expected skills of the readers data might be presented more formally using e.g., UML, or must definitely avoid any formalisms.
- *DMHA - Document Must Have Author:* Each software development document must explicitly list its authors, as in the case of changes each document has to be traceable to its creators. Therefore, this rule is violated, if there is no author defined in the document meta-information and no key word is found that indicates the existence of an author name.
- *DMHV - Document Must Have Version Id:* Similar to source code each document in a software project should have an explicit version identifier.
- *DMHVH - Document Must Have Version History:* In order to keep software development documents comprehensible, each document must provide a version history that roughly outlines the changes over time (versions) during the entire software development process.
- *DMS* - Document *Must have a State:* Each software development document should outline its defined state (e.g., draft, in review, final, customer approved), in order to present the current document state to the project members.
- *ECNF - Ensure Continuous Numbering of Figures:* In software development documents ascending numbering of figures improves the quality of documentation, as this contributes to a higher consistency and comprehensibility of the documents.
- *ECNT - Ensure Continuous Numbering of Tables:* In software development documents an ascending numbering of tables improves the document quality, as this leads to higher consistency and comprehensibility of the document.
- *EFRT - Ensure that each Figure is Referenced in the Text:* Each figure has to be referenced in the text of software development documents; otherwise it is incoherent or might be ambiguous.
- *ETRT - Ensure that each Table is Referenced in the Text:* Each table has to be referenced in the text of software development documents; otherwise it is incoherent or might be ambiguous.
- *FMHC - Figures Must Have a Caption:* Each figure in a software development document must have a caption in order to express the visualized topics linguistically; otherwise it may be ambiguous for the readers.
- *PIFF - Provide Index For Figures:* If a software development document contains figures, there must be an index listing all figures in order to keep information quickly retrievable for all project members.
- *PIFT - Provide Index For Tables:* If a software development document contains tables, there must be an index listing all tables in order to keep the information quickly retrievable for all project members.
- *TMHC – Tables Must Have a Caption:* Each table in a software development document must have a caption in order to express the visualized data of the table linguistically; otherwise it may be ambiguous for the readers.

C. Violations

In this section we give an overview of the results of our software development document defect detection analysis.

TABLE II. DEFECT DETECTION TOOL RESULTS

no. documents analyzed	50
no. document quality rules	24
total no. violations found	8,955
avg. false positive rate per rule	0.172

In our feasibility study 50 project documents were automatically checked by 24 document quality rules, which revealed a total number of 8,955 violations. For these findings we determined an average false positive rate per rule of 17.2 percent and a false negative rate per rule of 0.4 percent. False positive findings are (in our case) over-detected defects that are no documentation defects in the sense of human software inspectors. On the other hand, false negative findings are defects that have not been found by our tool but that are definitely documentation defects in the sense of human software inspectors.

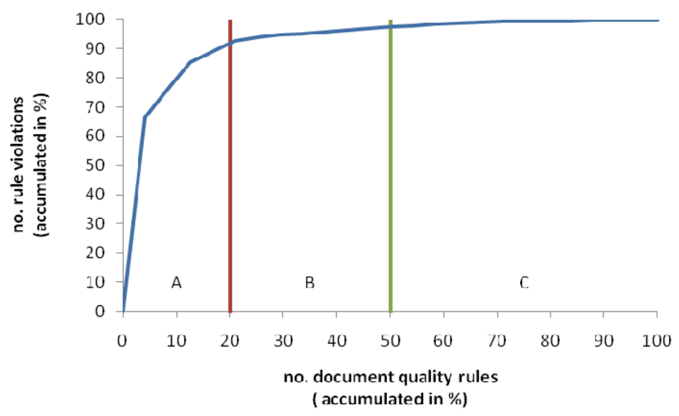

Figure 2. Rule violations per document quality rule distribution

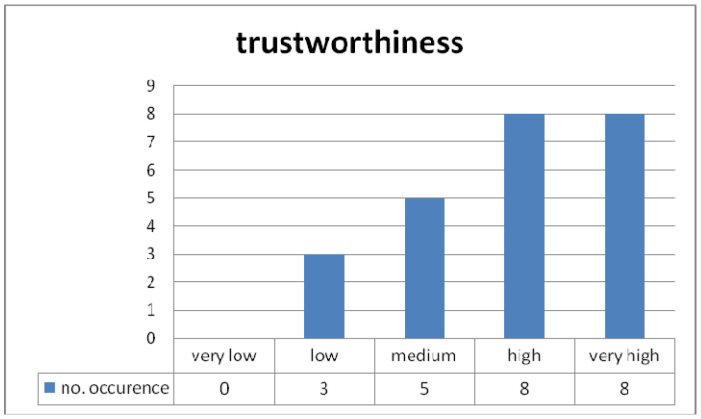

Figure 3. Trustworthiness of all applied document quality rules

TABLE III. RESULTS OVERVIEW PER RULE

rule	no. violations	false positive rate	false negative rate
ADNC	11	0	0
ADNS	14	0	0
ADUP	823	0	0
AES	337	0.033	0
AESD	30	0.933	0
AIDOC	0	0	0.020
ALS	49	0	0
AULD	9	0	0.100
ARHT	13	0.846	0
ASPE	5,956	0.602	0
ATSS	25	0	0
DESOR	50	0	0
DMHA	0	0	0.040
DMHV	21	0	0
DMHVH	14	0	0
DMS	48	0	0.040
ECNF	43	0.488	0
ECNT	46	0.326	0
EFRT	106	0.274	0
ETRT	75	0.160	0
FMHC	329	0.365	0
PIFF	50	0	0
PIFT	50	0	0
TMHC	856	0.093	0

During our investigations we also found out that the violations per rule are unequally distributed. As shown in Table III, the rules ADUP, AES, ASPE, FMHC and TMHC identified more than 300 violations each. Due to this, we accumulated the number of violations found by these five rules and compared it with the total number of violations. Consequently, as it can be seen in the ABC analysis diagram in Fig. 2, we revealed that these five document quality rules are responsible for more than 90 percent of all thrown violations.

D. *Trustworthiness*

The trustworthiness of a rule specifies how reliable the detection of a violation is. We classify trustworthiness into:

- *very low:* There is too much over- and/or under-detection in order to rely on the results.
- *low:* There is significant over- and under-detection.
- *medium:* Most issues are found, but there is over-detection.
- *high:* Almost no over- and under-detection. Very reliable findings.
- *very high:* No known over- or under-detection. Absolutely reliable findings.

As a result of this classification scheme, a main factor to determine the trustworthiness for a document quality rule is its false positive rate. Indeed, we also take false negative findings, as far as possible to indentify, and known weaknesses of the rule implementation into account, i.e., a rule with a false positive rate of 0.0 and/or a false negative rate of 0.0 does not implicitly have to have a trustworthiness rating of 'very high'.

As shown in Fig. 3, we rated the trustworthiness of the document violations for eight of our 24 applied rules with 'very high', i.e., these violations are very reliable.

TABLE IV. 'VERY HIGH' TRUSTWORTHY RULES

ADNC	Adhere to Document Naming Conventions
ADNS	Avoid Deeply Nested Sections
ADUP	Avoid Duplicates in Document
ALS	Avoid Long Sentences
ATSS	Adhere To Storage Structure
DMHVH	Document Must Have Version History
PIFF	Provide Index For Figures
PIFT	Provide Index For Tables

Furthermore, we also determined for eight document quality rules a 'high' trustworthiness, as we identified almost no over- or under-detection for this rules. As a result of this more than two-third of our rules are identified to be 'very high' or 'high' trustworthy.

TABLE V. 'HIGH' TRUSTWORTHY RULES

AES	Avoid Empty Sections
AIDOC	Avoid Incomplete Documents
AULD	Avoid Ultra Large Documents
DESOR	Define Expected Skills Of Readers
DMHA	Document Must Have Author
DMHV	Document Must Have Version Id
ETRT	Ensure that each Table is Referenced in the Text
TMHC	Table Must Have a Caption

However, our feasibility study also revealed three rules with a 'low' trustworthiness.

TABLE VI. 'LOW' TRUSTWORTHY RULES

AESD	Avoid Extremely Small Documents
ARHT	Avoid Repeated Heading Text
ASPE	Avoid Spelling Errors

These rules have to deal with a false positive rate of more than 60 percent, e.g. most of the ASPE violations are thrown as domain specific terms or abbreviations are falsely identified as misspelled words. Nevertheless, some of the violations of these three rules are informative as we think that, although there is much over- and under-detection, they can be categorized as 'low' trustworthy. Moreover, we think that small rule improvements, e.g. adding the usage of a domain specific dictionary for the ASPE rule, would lead to a higher trustworthiness.

E. *Effort to fix defects*

The effort to fix true positive findings specifies how much is needed to spent for removing a defect (qualitatively):

- *low:* Only some local lines in a document have to be changed.
- *medium:* Document-wide changes are necessary.
- *high:* Project-wide document changes are necessary.

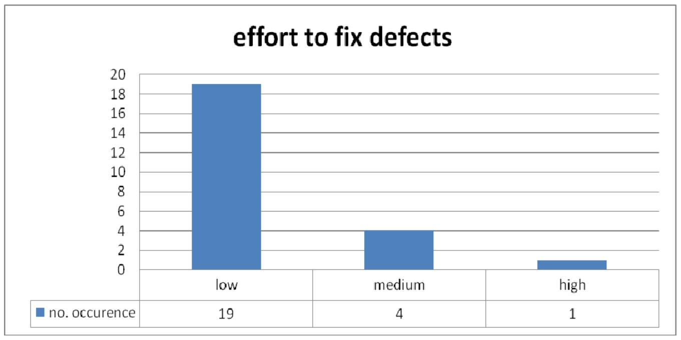

Figure 4. 'Effort to change defects' of all applied document quality rules

As shown in Fig. 4, most violations thrown by 19 of our 24 applied rules affect only some lines in the documents, i.e. these defects can be quickly corrected and represent easy wins. Moreover, for fixing the defects of four of our rules we determined that document-wide changes are required.

TABLE VII. 'MEDIUM' EFFORT TO FIX DEFECTS

ADNS	Avoid Deeply Nested Sections
ADUP	Avoid Duplicates in Document
AESD	Avoid Extremely Small Documents
DMHVH	Document Must Have Version History

Nevertheless, during our feasibility study we also determined that all true positive AULD violations lead to project-wide document changes. In this case, high effort is needed as an ultra large document has to be split into separate documents. Furthermore, all references are affected and have to be checked for correctness. It is very hard to determine whether defects of a specific rule generally affect only some lines in a document or the entire software project, as e.g. small changes in some lines can also lead to broken references in other documents.

F. *Effort to change settings*

The effort to adapt configuration settings of the rules to the needs of the specific project specifies how much effort is needed to spent for adapting the rule configurations, before the document defect detection tool can be applied:

- *low:* Nothing or very small adaptations are necessary in a settings file.
- *medium:* Some lines have to be changed in a settings file. Some knowledge of the analyzed project documents is necessary to define, e.g., suitable regular expressions.
- *high:* Settings files have to be changed considerably. Detailed information of the project document content and structure is necessary to define, e.g., suitable regular expressions.

As stated in Fig. 5, more than two-thirds of all applied document quality rules do not need considerable effort to be

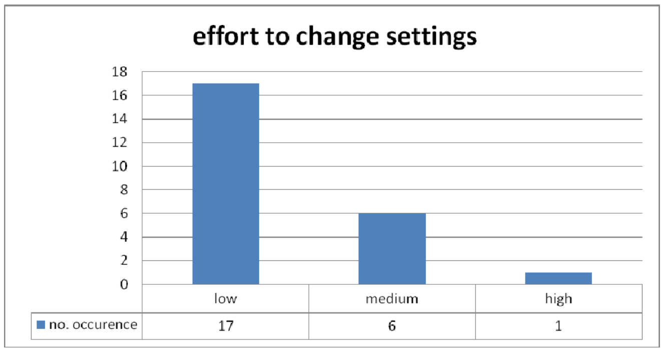

Figure 5. 'Effort to change settings' of all applied document quality rule

suitably configured. In order to correctly configure six of our rules it is necessary to have some further knowledge in specifying correct regular expressions.

TABLE VIII. 'MEDIUM' EFFORT TO CHANGE SETTINGS

ADNC	Adhere to Document Naming Conventions
ASPE	Avoid Spelling Errors
ATSS	Adhere To Storage Structure
DMHV	Document Must Have Version Id
EFRT	Ensure that each Figure is Referenced in the Text
ETRT	Ensure that each Table is Referenced in the Text

Furthermore, it is required to have an overview of the document structure and document content. However, to correctly configure the DESOR rule (effort to change settings = 'high'), there must be also some knowledge of the used expressions and languages in order to identify and extract the specific document content properties that define the skills of readers.

VI. CONCLUSION AND FURTHER WORK

Empirical studies show that tool support can significantly increase the performance of the overall software inspection process [10][11][12][13]. However, most available software inspection tools are optimized for code inspections, which usually provide support for plain text documents, only. Due to this they are inflexible with respect to different artifact types and limit inspectors in their work. For natural language text, inspection tools cannot fully replace human inspectors in detecting defects. Nevertheless, software inspection tools can be used to make defect detection tasks easier [11]. Encouraged by this, we developed a tool-based quality defect detection approach to support the inspection process by checking documentation best practices in software development documents.

International documentation and specification standards [22] [23] [24] [25] [26] define a set of generally accepted documentation best practices. Furthermore, checklists and reviewing procedures [27] are widely used as well as documentation quantifiers in order to check specific documentation characteristics representing quality aspects [28]. As a result of these studies we came to the conclusion, that measurable document quality rules expressing best practices can also be used to detect defects in software development documents and to help enhancing documentation quality. So far we have implemented a document quality defect detection tool, which is applicable on Office Open XML documents [29] and Microsoft Office Binary Format documents [31]. The tool allows checking, whether software development documents adhere to explicitly defined document quality rules. In a feasibility study we showed that our automatic defect detection tool is capable of finding additional uncovered significant documentation defects that had been overlooked by human inspectors.

During our analysis, we automatically checked 50 Microsoft Office Word documents of a real-world software project with 24 document quality rules. The tool revealed 8,955 violations with an average false positive rate per rule of 17.2 percent and an average false negative rate per rule of 0.4 percent. As our study shows, two-thirds of all applied document quality rules were rated with a 'high' or 'very high' trustworthiness. Furthermore, it has been pointed out that most of the violations found can be easily removed (effort to change defect = 'low'), as they often only affect some few lines.

In our feasibility study we determined that nearly 75 percent of all rules did not need any further configuration changes before they could be suitably applied to software development documents. Nevertheless, seven rules had to be adapted to project specific document conventions before they could be applied. In the case of the project documentation used for our study the configuration of these rules took us approximately six hours, as we were not familiar with the conventions defined for the document naming and content structure. However, we saw that after the rules had been suitably configured, the trustworthiness of the rule violations rose considerably, i.e., the configuration effort well paid-off.

In a next step, we will apply our document quality defect detection tool on the documents of additional software projects to improve the implementation of the rules with an emphasis on reducing the false positive rate and to validate the results of our feasibility study in more detail. Moreover, as we have seen that some of our rules are applicable for most technical documents, we also want to implement some document quality rules that are even more specific for software development rules. For instance, we will add rules that deal with domain specific terms and glossaries used in software documents or the traceability of references between various software development documents (of different software life-cycle phases).

We currently also work on transferring Adobe PDF documents in a way that the already developed document quality rules for the Office Open XML documents and Microsoft Office Binary Format documents can be used without changes. As a result of this, we think that the definition of an abstract document structure that separates the rules from the underlying software artifacts is essential. Consequently, this would enable a much easier development of rules that can be applied on elements of a general document model, as there is no need to deal with the complexity of specific document formats for the rule development. Furthermore, we recognized that our rules are too loosely grouped. From our experience with rules in the context of code quality [35], we will develop a quality model that allows a systematic clustering of rules by means of quality attributes. This will give us the possibility to evaluate document quality on more abstract levels, like readability or understandability of a document.

ACKNOWLEDGMENTS

We would like to thank Siemens AG Corporate Technology for supporting our empirical investigations by providing us with software development documentation data in order to conduct our feasibility study and test our approach.

REFERENCES

[1] B. W. Boehm, Software Engineering. Barry W. Boehm's lifetime contributions to software development, management, and research. Hoboken, N.J., Wiley-Intersience, 2007.

[2] L. Briand, K. E. Emam, O. Laitenberger, and T. Fussbroich, Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects. International Conference on Software Engineering, IEEE Computer Society, 1998.

[3] J. C. Chen and S. J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," in Journal of Systems and Software. Elsevier Science Inc., 2009, vol. 82, pp. 981-992.

[4] M. E. Fagan, "Design and code inspections to reduce errors in program development," in IBM Systems Journal. vol. 15 (3), 1976, pp. 182-211.

[5] T. Gilb and D. Graham, Software Inspection. Addison-Wesley Publishing Company, 1993.

[6] T. Thelin, P. Runeson, and C. Wohlin, "An Experimental Comparison of Usage-Based and Checklist-Based Reading," in IEEE Trans. Software Engineering, vol. 29, no. 8, Aug. 2003, pp. 687-704.

[7] T. Thelin, P. Runeson, C. Wohlin, T. Olsson, and C. Andersson, "Evaluation of Usage-Based Reading-Conclusions after Three Experiments," in Empirical Software Engineering: An Int'l J., vol. 9, no. 1, 2004, pp. 77-110.

[8] F. Shull, I. Rus, and V. Basili, "How Perspective-Based Reading Can Improve Requirements Inspections," in Computer, vol. 33, no. 7, July 2000, pp. 73-79.

[9] J. Carver, F. Shull, and V.R. Basili, "Can Ovservational Techniques Help Novices Overcome the Software Inspection Learning Curve? An Empirical Investigation," in Empirical Software Engineering: An Int'l J., vol. 11, no. 4., 2006, pp. 523-539.

[10] O. Laitenberger and J.-M. DeBaud, "An encompassing life cycle centric survey of software inspection," in Journal of Systems and Software. vol. 50, 2000, pp. 5-31.

[11] S. Biff; P. Grünbacher, and M. Halling, "A family of experiments to investigate the effects of groupware for software inspection," in Automated Software Engineering, Kluwer Academic Publishers, vol. 13, 2006, pp. 373-394.

[12] H. Hedberg and J. Lappalainen, A Preliminary Evaluation of Software Inspection Tools, with the DESMET Method. Fifth International Conference on Quality Software, IEEE Computer Society, 2005, pp. 45-54.

[13] V. Tenhunen and J. Sajaniemi, An Evaluation of Inspection Automation Tools. International Conference on Software Quality, Spinger-Verlag, 2002, pp. 351-362.

[14] W. M. Wilson, L. H. Rosenberg, and L.E. Hyatt, Automated quality analysis of Natural Language Requirement specifications. PNSQC Conference, October 1996.

[15] G. Lami and R. W. Ferguson, "An Empirical Study on the Impact of Automation on the Requirements Analysis Process," in Journal of Computer Science Technology. vol. 22, 2007, pp. 338-347.

[16] G. Lami, QuARS: A tool for analyzing requirements. Software Engineering Institute, 2005.

[17] P. Jain, K. Verma, A. Kass, and R. G. Vasquez, Automated review of natural language requirements documents: generating useful warnings with user-extensible glossaries driving a simple state machine. Proceedings of the 2nd India software engineering conference, ACM, 2009, pp. 37-46.

[18] Raven: Requirments Authoring and Validation Environment, www.ravenflow.com.

[19] T. Farkas, T. Klein, H. Röbig, "Application of Quality Standards to Mutliple Artifacts with a Universal Compliance Solution", in Model-Based Engineering of Embedded Real-Time Systems. International Dagstuhl Workshop, Dagstuhl Castle, Germany, 2007.

[20] Microsoft Developer Network: The LINQ Project, http://msdn.microsoft.com/en-us/netframework/aa904594.aspx

[21] J. Nödler, H. Neukirchen, and J. Grabowski, "A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications" in Proceedings of the 2009 International Conference on Software Testing Verification and Validation. IEEE Computer Society, 2009, pp. 101-110.

[22] NASA Software Documentation Standard, NASA-STD-2100-91. National Aeronautics and Space Administration, NASA Headquarters, Software Engineering Program, July, 1991.

[23] IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998. 1998.

[24] IEEE Standard for Software User Documentation, IEEE Std 1063-2001. 2001

[25] ISO/IEC 18019:2004: Software and system engineering - Guidelines for the design and preparation of user documentation for application software, 2004.

[26] ISO/IEC 26514:2008: Systems and software engineering - Requirements for designers and developers of user documentation, 2008.

[27] G. Hargis, M. Carey, A. K. Hernandez, P. Hughes, D. Longo, S. Rouiller, E. Wilde, Developing Quality Technical Information: A Handbook for Writers and Editors, 2nd ed. IBM Press, 2004.

[28] J. D. Arthur, and K. T. Stevens, Document Quality Indicators: A Framework for Assessing Documentation Adequacy. Virginia Polytechnic Institute, State University, 1990.

[29] ISO/IEC 29500:2008. Information technology – Document description and processing languages – Office Open XML File Formats Open Document Format, 2008.

[30] ISO/IEC 26300:2006. Information technology- Open Document Format for Office Applications (OpenDocument), 2006.

[31] Microsoft Office Binary File Format: http://www.microsoft.com/interop/docs/OfficeBinaryFormats.mspx

[32] P. Buchlovsky and H. Thielecke, "A Type-theoretic Reconstruction of the Visitor Pattern. Electronic Notes," in Theoretical Computer Science. vol. 155, 2006, pp. 309 - 329.

[33] B. C. Oliveira, M. Wang, and J. Gibbons, The visitor pattern as a reusable, generic, type-safe component. Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. ACM, 2008, pp. 439-456.

[34] T. Copeland, PMD Applied. Centennial Books, 2005.

[35] R. Plösch, H. Gruber, A. Hentschel, Ch. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck, "The EMISQ Method and its Tool Support - Expert Based Evaluation of Internal Software Quality," in Journal of Innovations in Systems and Software Engineering. Springer London, vol. 4(1), March 2008.