January 2014

Software Documentation Guidelines For Maintainability

Justin Roger Huber

SOFTWARE DOCUMENTATION GUIDELINES FOR MAINTAINABILITY

by

Justin Roger Huber
Bachelor of Science, Dickinson State University, 2009

A Thesis
Submitted to the Graduate Faculty

Of the

University of North Dakota

In partial fulfillment of the requirements

For the degree of

Master of Science

Grand Forks, North Dakota
May
2014

This thesis submitted by Justin Roger Huber in partial fulfillment of the requirements for the Degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

Dr. Hassan Reza

Dr. Travis Desell

Dr. Wen-Chen Hu

This thesis is being submitted by the appointed advisory committee as having met all of the requirements of the School of Graduate Studies at the University of North Dakota and is hereby approved.

Wayne Swisher
Dean of the School of Graduate Studies

May 2, 2014
Date

PERMISSION

Title Software Documentation Guidelines for Maintainability

Department Computer Science

Degree Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work or, in his absence, by the Chairperson of the department or the dean of the School of Graduate Studies. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Justin Roger Huber
5/1/2014

TABLE OF CONTENTS

LIST OF FIGURES ……..... vi

ACKNOWLEDGEMENTS …….... vii

ABSTRACT viii

CHAPTER

I. INTRODUCTION ……. 1

II. BACKGROUND ……... 4

Maintainability ……... 4

Types of Change …… 6

Software Architecture Maintainability …….. 7

Maintenance Tactics ……..……..……..... 10

Maintainability Evaluation Methods ……...……. 12

Software Documentation ……..……... 14

III. METHOD ……..…….. 20

Software Documentation Tactics ……...…….. 21

Application of Software Documentation Tactics ……...…….. 30

Case Study ……..……. 35

IV. DISCUSSION ……...……... 48

Future Application of Method ……..……... 48

Conclusion ……..……. 54

REFERENCES ……..…….. 58

LIST OF FIGURES

Figure Page

1. Poor Requirement Example ..…….…..…..…. 23

2. Poor Test Example .…….…..……. 24

3. Design Example …….…..…..…..…. 25

4. Source-code Example 27

5. Documentation Cycle ….…….…..…..… 29

6. Cohesion & Coupling - Requirement …..….……..……..……. 37

7. Cohesion & Coupling - Test 37

8. Size & Understandability - Requirement .....……. 38

9. Size & Understandability - Test .....……..…….....……. 39

10. Traceability & Networking - Requirements 42

11. Traceability & Networking - Design …...……..……..……. 43

12. Traceability & Networking - Testing .....……..……..... 45

13. Traceability & Networking - Source-code 46

14. Evaluation Example .....….…….…..…..... 49

15. Blackboard Solution Example …….....….. 53

16. Related Works ……..... 56

ACKNOWLEDGEMENTS

First and foremost I wish to express my sincere appreciation to my thesis advisor, Dr. Hassan Reza, for his guidance, support, and encouragement during my time in the master's program at the University of North Dakota. The knowledge he has provided will be invaluable in my future endeavors in software engineering. I would also like to thank the members of my thesis advisory committee, Dr. Wen-Chen Hu and Dr. Travis Desell, and my teaching assistantship supervisor, Mr. Scott Kerlin, for their teachings and support. To my fellow graduate teaching assistants, I thank you for your support, knowledge, and companionship these last two years. Finally I would like to thank my family for their love and support while on this academic journey.

To Lauren E. Payne,

I dedicate this master's thesis to you. Your everlasting love and support continues to push me toward achieving my goals and untapped potential. Without you this would not have been possible.

I love you.

ABSTRACT

Software maintenance is the final stage in the software engineering process; and can also be the longest and most costly. The ability of, and ease in which software upkeep and change is performed is an important software quality attribute called maintainability. Software maintenance however is an expensive process in both time and financial cost and can account for a majority of a system's overall lifetime cost. It would be beneficial then to have a set of guidelines for obtaining a higher degree of maintainability in software before and after its release and evolution stage in order to limit the cost of change. Six suggested steps for achieving a higher degree of maintainability through software documentation are provided along with a short case study, followed by a discussion on potential results and future enhancements to the proposed method.

CHAPTER I

INTRODUCTION

Change is an inevitable and necessary event. From climates, down to landscapes, people, animals, plants, and microorganisms, all evolve to survive in a dynamic world. A software system is no different, and must adapt to meet the ever-changing needs of its operating environment, stakeholders, and users. In order to satisfy these needs, software must be maintained, modified, and updated from the design, development, release, and post release phases. Change does not come cheap however, as making changes to software costs time, resources, and funding. Information systems, for example, can use up to 80% of their budget on maintenance costs alone [39]; and over any software's lifetime, the cost of change and upkeep can amass to at least 50% of the total system cost [9] [45].

It has been shown that design decisions have an impact on maintenance concerns and can have long lasting and irreversible effects on the software [5]. Thus maintainability has become an important consideration when designing any software system and there is a rise in programmers whose focus is primarily on maintenance. According to Capers Jones [44] since 2000 the number of programmers in maintenance will have risen 50% from 6 to 9 million and is projected as 67% of the total programmer work force. This 50% increase trend is shown to continue in the next 10 years. Legacy

systems are a big contributor to this statistic as the cost of maintenance compared to bringing the system up to date is more cost effective [44]. Statistics like these show that a balance should be struck between system cost and change. As the difficulty of implementing change rises, so too does cost, as the system will require more resources and development time.

It is clear that a method for achieving a higher degree of software maintainability is crucial to not only the product's success, but a company's bottom line as well. A building architect knows that the ability for a structure to be changed or expanded upon starts at the foundation, its architecture. If the design does not accommodate change well, it becomes increasingly difficult, possibly impossible, for updates and maintenance to take place; and the same can be said for any software system. Current solutions for evaluating software, such as the Architecture Tradeoff Analysis Method (ATAM) [29] and the Software Architecture Analysis Method (SAAM) [30], focus on the architecture and design of the system; showing that quality attributes, such as maintainability, start at the architecture and design phase in the software lifecycle. Scenario generation and analysis is used to determine, in a qualitative way, how well architecture adheres to a given software quality attribute. Methods such as these work in the general sense, where a system can be analyzed based on any software quality attribute, such as performance or availability, but do not focus on one single attribute, nor do they focus on all the components that affect the chosen attribute. A more focused evaluation method has the potential for uncovering more information about the desired quality of a system at different levels depending on the attribute chosen. Maintainability, for example, can be affected not only by the software's design, but by its documentation, which starts at the

requirements and specification stage of the development cycle.

This paper seeks to uncover that potential by proposing techniques for increasing maintainability in a software system through the documentation from four phases of the development cycle starting with requirements, to design, code, and finally testing. These four stages continue long after the release of software and are intertwined in a cycle of documentation change alongside system change. Six guidelines: cohesion, coupling, size, understandability, traceability, and networking; are presented along with a case study to show their application. While the first five guidelines are commonly associated with different levels of maintainability, as described in this paper, the sixth guideline, networking, appears underutilized. This paper seeks to add and show how the external networking, or referencing, of documentation can provide a greater degree of consistency and ease in maintenance execution and understanding. Suggested future applications of the presented method are also discussed.

The remainder of this paper is organized as follows. Chapter II provides a background of software maintenance, maintainability through software architecture, and maintainability through software documentation. Chapter III will present a new method for evaluating maintainability, followed by a hypothetical case study. Finally, future applications, conclusions and current issues or open research questions for the method are provided in Chapter IV.

CHAPTER II

BACKGROUND

Maintainability

In software engineering two common types of requirements described are functional and nonfunctional [6] [45] [7] [22]. Functional requirements detail what a system should be doing and how it should behave. Functional requirements can be described through many different means, such as requirements documentation, use cases, and user manuals to name a few. Nonfunctional requirements are a way of measuring the system behavior, based on the functional requirements. Nonfunctional requirements are detailed through the software architecture and are also known as quality attributes. Some common quality attributes include performance and availability [6] which are concerned with, but not limited to, speed and uptime functional issues respectively. There are many other software quality attributes used to describe software systems, but one, modifiability, is closely related [11] to maintenance.

Modifiability is about change, the cost of change, and the probability of change [6]. Modifications can range from simple code or design changes, such as feature optimization, addition, and deletion, to complete system overhauls such as changing to a more accommodating framework, database, or architecture. These types of change, and more, are explained in the next section. Cost can be split into two types, financial and

time [4]. Financial costs are accumulated through labor and hardware or software changes. The more people working on a change, the higher the cost of that change rises. Some modifications may need additional hardware or software which incurs their own additional financial cost. Time costs are accumulated through the time it takes to complete a change, and financial costs from labor are a direct consequence of time. The longer a modification takes to discover, implement, test, and document, the higher the cost of that change; and time and resources could have been spent on other projects. Hardware or software changes, for example, in some cases, may require additional training which adds to time cost. The person-hour is used to correlate and determine these time and financial costs. For the purposes of this proposal, the term updatability has been generated to describe a quality attribute defining the ease in which modifications can be implemented successfully. Difficulty of change is attributed to manpower and is correlated with time cost. As the difficulty of modifications rise, so too do the number of workers required, and subsequently, the number of person-hours. Finally, modifiability is also concerned with finding the probability of change. Change scenario difficulty and the likelihood they will or will not happen are important in determining if system design alterations should be made in order to accommodate more high-risk modifications.

Combining these two quality attributes, a more detailed definition of maintenance can be derived and stated as: the capability of, or ease in which, a software product can: improve on performance or other quality attributes, add new features or improve on old ones, be reused (with modifications), adjusted to changing requirements and environments, or fault corrected; all of which are called for by either stakeholders, customers, or the operating environment. Maintainability, then, can be used to describe

both of these properties and can be defined as the cost and capability of a software product to adapt to meet the changing needs of stakeholders, customers, and environments.

Types of Change

At its core, maintainability is about change and sustainability. The software development process [45] ends on a continuing cycle of upkeep and change from both internal and external sources. As mentioned prior, there are many types of change that range from simple bug fixes, to entire system restructuring. Every modification is different and unique in its own right, however, they can be categorized together by similar traits in order to more easily identify how best to analyze and implement them. There are four categories of change [30], which were derived from [40]. From information provided in [30] [12], these categories will be listed below and described in more detail in the paragraphs that follow.

1. Capability Extension: Addition or enhancement of features
2. Capability Deletion: Removal of features
3. Environment Adaptation: Changes in hardware or environment software
4. Restructuring: Optimization and reuse of components and services

Any additional features to be added, or enhancements made to current features are categorized under the first change category, capability extension. Feature addition is, for the most part, going to be dictated by the needs of the customer, or customer base, the software was intended. The need for a new feature could be caused by the necessity to keep up with competing software, changes in business practice (internal or clientele), or

to fulfil requirements that were missed on the initial release, which is common in agile development. These types of change would typically be reserved for software expansions, or updates. Feature enhancements are about making additions to, or fixing problems to currently implemented features. This is where most bug fixes will be categorized and where the general term, maintenance, most closely fits. Additions to features are those extensions that add something, or enhance current features such as adding a sorting function to a report or list, or adding an extra field to a form. These types of change are more easily implemented and would typically be included in a smaller updates such as hotfixes. Capability extensions will most likely be the easiest to implement as they are focused on particular functions rather than parts of the design itself. While there could be some overlapping change with other features, for the most part these types of change are isolated.

The removal of features is categorized under the second type of change, capability deletion. Like capability enhancement, capability deletion is dictated by the clientele, but can also be driven by the development team. Over time, features can go unused, become unwanted or, in the case where a lot of change is focused on one feature, a hindrance to development. If a customer no longer needs a feature, or it becomes apparent that a feature is no longer relevant, a removal request may be put in. Features that continually require maintenance may also be determined as a hindrance to development, in which case they could be deleted and/or re-implemented. This type of change is a little more difficult than the first type, as a feature may be coupled to other important or working functions of the software. Removing a feature completely may require changes to many other functions or components which drive the cost of the change up. The feature may

also be desired again at a future date; in these cases simply hiding it may be a better option. Components-off-the-shelf (COTS) or product families are examples of software that make frequent use of this type of change in conjunction with the first type for different versions of the software for different customers.

The third type of change, environment adaptation, is about changing the system to meet the demands of the environment in which the software is implemented. These types of change are either hardware or software driven. Hardware environment changes involve updating the software to perform on a different hardware than originally intended, such as becoming compatible on mobile platforms where it was originally designed for use on a PC. Software environment changes involve updating the software to perform with other software than originally intended, such as becoming compatible with the Linux operating system where it was originally designed for use on a Windows platform. This category of change is much more involved than the first two, and simply making changes to the current version of the software may be impossible; in this case a new, sister version may be created, which uses similar features and functions but updated to work with different hardware or software.

The fourth and final type of change is restructuring. Restructuring is about changes that affect quality attributes, such as performance, to either individual features or the system as a whole; as well as changes to system components or features for reuse in other software, or changing the current software to use reusable system components or features. The difficulty of quality attribute changes is largely dependent on the nature of the change. Take a performance enhancement for example, if it is simply a change in how the code is structured in order to speed up the response time of a particular feature, the

implementation could be relatively simple with only a moderate time investment; however if the change requires that the system as a whole becomes more responsive, this could require a change to the architecture, which could be a massive undertaking. Maintenance for implementing reusable components or features, such as COTS, is useful when the cost of creating features from scratch outweighs the benefits and cost of using reusable software which may have already established documentation and support.

The preceding paragraphs gave only a summary of what each type of change offers. There are many change requests that can be generated for all types of software, and each type of change comes with its own documentation and is affected differently by the change request, software, requirements, and design. Software architecture and documentation play an important role in change and are discussed next.

Software Architecture Maintainability

Software architecture is the "set of principal design decisions made during development and subsequent evolution" [48]. These design decisions include what style, or framework, the system will be built under, such as an object-oriented, layered, or client-server style, among many others [43], as well as quality attributes such as maintainability, performance, availability, security, and more. Selecting and building from a proper architecture based on the requirements of the software can help ensure a smoother development cycle and future success post-release. Decisions made during the architecture design have long lasting impressions on a desired quality attribute and certain architectural styles are better suited for different sets of qualities. The four types of change are each affected differently depending on the architecture. An object-oriented

architectural design, for example, is well suited for maintainability, in general, as it allows for a high level of cohesion and low coupling between classes which provides greater ease in types one and two. Layered architecture, on other hand, can also provide sufficient maintainability however since each layer can affect the layer above and/or below it there is the potential for more coupling than desired compared to an object-oriented architecture. The following sub-sections detail characteristics of maintainability in architecture, such as cohesion and coupling, as well as current known methods for evaluating for software maintainability.

Maintenance Tactics

If nonfunctional requirements, software quality attributes, are used to measure system behavior set by functional requirements, then tactics [6] are used to measure nonfunctional requirements. Tactics, or metrics, are the techniques used to attain a desired quality attribute in software architecture, with each attribute containing its own set of tactics. Primarily, there are three tactics for maintainability: coupling, cohesion, and size. Each tactic is used for determining maintainability in documentation, architecture, and implementation.

Coupling is used to describe how different aspects of the system, internal or external, are linked. If one item of the system is linked, directly or indirectly, with one or more items, these are said to be coupled. Maintainability is concerned with how these items are linked by change, not function. For example, in speaking of architecture, if making a change to one system component requires a subsequent change in one or more other components, they are coupled. As coupling between components, documentation,

code, etc. rises, so too does maintenance, which lowers maintainability.

Cohesion is used to describe how different aspects of the system are focused on the function given to them; if a system component was designed to perform a specific function, it should operate only within that function definition. The more focused a component is, the higher the cohesion. Object-Oriented architectures provide a good example for highly cohesive systems. Classes are built in such a way that their contents relate only to the definition of the class set by the requirements. Cohesion is important to maintainability in that as the level of cohesion in system components rises, the coupling between them tends to drop. For example, with a high level of cohesion, even if changing a piece of one component requires changing other pieces within that component, the need to change other, separate, components in the system is reduced which decreases the difficulty of maintenance.

Size is important to maintainability in terms of cost. As the size of the program, modules, system components, and code increase so too does the amount of time it takes to make changes. Increased time cost, as discussed earlier, will bring rise to financial cost. A smaller program will have fewer components and, ideally, less code than a larger program which will take less time and personnel to complete modifications. SIZE1 and SIZE2 [9] are maintainability metrics used to describe size of code. SIZE1 is the total lines of code in a class or component, where SIZE2 is the total number of properties, or fields, plus the total methods. Size is affected by a number of factors, such as developer knowledge, as well as coupling and cohesion. As the knowledge and talent of a single developer, or team of developers, increases, the impact of size is reduced. A highly cohesive and lowly coupled set of components will also be easier to maintain.

Maintenance tactics are an important tool for measuring maintainability in software systems. The examples above described how tactics affect the software in low-level architecture and implementation. Maintainability tactics also play a role at the documentation level and are further discussed at the end of this chapter.

Maintainability Evaluation Methods

Since software architecture design occurs so early in the life-cycle, it is important to detect issues with nonfunctional requirements before determining too deep into development that the software does not meet them. For those systems that require a high level of maintainability, a platform for which to evaluate this quality is important in determining the success of the architecture in meeting the requirements. If maintainability is a desired attribute, but partway through development, or post release, the system is found to be difficult to maintain, the option of going back and changing the architecture can be next to impossible due to funding and resources. There are many methods currently available for evaluating software architectures, two of them are discussed in the proceeding paragraphs. The methods are ordered as follows: the Software Architecture Analysis Method, and the Architecture Tradeoff Analysis Method.

The Software Architecture Analysis Method (SAAM) [30] is a, scenario based, software architecture evaluation method whose goal is to determine if the architecture fits a desired quality attribute, given a set of testable scenarios. The method is built so that any quality attribute can be chosen for evaluation, given a proper set of scenarios and system component definitions. SAAM uses three perspectives for evaluating the architecture: functionality, what the system does, structure, components and connectors,

and finally allocation, how the function is implemented in the structure. SAAM provides a good base for evaluating architectures for quality attributes and allows for any desired quality attribute to be interchanged within the method in order to provide an analysis on said attribute and architecture. SAAM has been demonstrated using modifiability as the test, and been shown it to be effective [30]. The issue with this method however, is that it lacks focus and does not give any quantifiable data, or metrics, representing the suitability of a specific quality of the software architecture, such as maintainability. For a more detailed analysis of specific attributes, enhancements would need to be made.

While SAAM provides a sufficient method for evaluating one chosen architectural quality, it isn't concerned with the tradeoffs from choosing one attribute over another. It is said that every action has an equal and opposite reaction; the same can be said about design decisions in software architecture. The Architecture Tradeoff Analysis Method (ATAM) [29] was created to find these opposite reactions, or tradeoffs, when choosing a particular quality attribute in the software design. There are those quality attributes that share certain desirable effects on the system, and there are those that trade off from one another, such as having higher availability also means that security must be increased, or having higher complexity reduces ease of modifiability. Understanding and recognizing these tradeoffs is paramount during architectural design as there could be several desirable quality attributes needed for the system, yet 2 or more negatively impact the other. A balance must then be struck in order to reach the goals set forth by the requirements and architecture. The ATAM aims to attain this balance and provides another useful way of evaluating software architectures for desired quality attributes. However, like the SAAM it does not provide a means for quantitatively

measuring a quality attribute and it is more suited for instances where once a primary quality has been achieved, such as maintainability, the secondary qualities can be evaluated against it and adjustments can be made if necessary.

The preceding architecture evaluation methods are just a small sample of what is available. There are many other software architecture evaluation methods [1], some that fit multiple quality attributes, such as the Quality Attribute Workshop (QAW) [10], a lighter version of the ATAM, and some that have been shown to qualitatively and effectively evaluate for maintainability, such as the Quality-driven Architecture Design and Analysis Method (QADA) [36]; and many more that are beyond the scope of this paper.

The information presented in this section provided a background of research for maintainability in software architecture. The method to be presented next, however, seeks to further maintainability through documentation, not architecture. There are changes that have an insignificant impact on the overall design and structure on the system, and then there are those that require structural change. For the purposes of this paper, these types of change will be called architecture insignificant, and architecture significant change respectively. The remaining section of this chapter discusses different levels of software documentation followed by a proposed method for achieving a higher degree of maintainability in chapter III.

Software Documentation

The previous section described how the early design decisions, or architecture, can have long lasting effects on system maintainability. However, an even earlier stage in

the software engineering process, requirements and specification elicitation and documentation, where the functionality is defined and documented, can be just as impactful. Software creation starts and ends with documentation, beginning with the requirements and ending in a repeating cycle of maintenance and testing. Detailed and concise documentation throughout the development process is a key component [19] to the longevity and maintainability of software; and lack of proper documentation has been shown [47] to make a significantly negative effect on the maintenance quality and success of software. Documentation is the blueprint for which designers, programmers, and testers use to make sure that software maintenance requests are implemented, or discarded, correctly. It is important that along with the architecture and design, the documentation be outfitted for software maintainability as well. All software development processes, whether they are specification or evolutionary based [46], use documentation in some form and at different frequencies. Development processes using the waterfall model [42], for example, are far more reliant on strong and large quantities of documentation than agile development methods. Two terms will be used in the rest of this paper and their distinction is defined before continuing with the discussion on documentation. A document will refer to the overall grouping of information presented by the different document types, while an artifact will refer to individual items, or articles, within the document, or supplementing materials for the document. The following paragraphs detail the four areas of documentation that will be used in this paper: requirements, design, code, and testing.

Requirements and specification elicitation is the first stage of the software engineering process. The requirements document is a form of external documentation that

defines what the software is and all of its proposed functions that are required to meet the needs of the client requesting the software [45] [22]. It can include, but need not be limited to: system functions and expected behaviors; specific hardware and software requirements; system, hardware, or software constraints; glossary of terms; and project goals and customer expectations. Software architecture and design implement the functionality set by the requirements, so while they affect maintainability at a structural level, the requirements affect maintainability at a conceptual level. Vague or incomplete requirements can result in a poorly implemented architecture [47] which inevitably will not only drive up maintenance requests, but costs as well. The ease in which maintenance is determined to be needed or discarded is also related to the quality of the requirements. For example, capability extension or deletion (which includes bug or error fixes) requests can be determined invalid or valid based on functionality defined in the requirements. An invalid request would show that the software is working as intended or there was user error, while a valid request would show that the implementation of the requirements was incorrect, or a beneficial addition to the requirements can be considered. Detailed requirements can also validate the need for the fourth type of change, restructuring, as the documentation should also include functional behavior of the system.

Architecture, or design, documentation describes how the requirements laid out in the requirements document will be realized and implemented. While requirements detail the system at the conceptual level, design documentation details the system at the structural level. The software design document (SDD) provides guidance to the development team on the architecture of the developing product. Architectural design is split into two levels: high-level design and low-level design. High-level design

documentation attempts to describe the design of the system in easy to understand modules representing interfaces and relationships among components as well as the architecture style that will be employed. Low-level design documentation provides more details for the modules and relationships provided by the high-level design. Algorithms, data flows and structures, class diagrams, and more are provided by low-level design. The SDD is often split into four categories: data design which describes the data structure and relationships among different data, architecture design which maps the program structure and framework, interface design which describes the program interfaces both internal and external, and finally procedural design which describes programming structure for translation to code. The IEEE 1016 standard for the SDD employs these categories [25]. Modeling documentation languages such as the Unified Modelling Language (UML) [48] and the Architecture Analysis and Design Language (AADL) [16] are also branches of software design documentation and provide visual documentation along with textual. Software design documentation is important to the maintainability of the system as it provides the proposed solution to the requirements and guidelines for implementation of code. A poorly detailed SDD or its supporting artifacts may cause a misunderstanding of what is needed to implement the requirements, resulting in an incorrect system.

Where the architecture and design implements the requirements, code implements the architecture and design. Source code documentation is a form of internal documentation that describes the intention of code, or its implementation of architecture. Documentation includes, but needs not be limited to, comments on individual lines of code, functions, and classes. Code documentation can be invaluable to maintenance as it

has the potential to make the worst code somewhat manageable, and increase understandability in the best code. Code is maintained by many different people and can often be by someone other than who wrote it; without proper documentation, any of the types of change become difficult as different programmers may not understand the way the code is written and what parts it affects, or the original writer may have forgotten his, or her, intent after coming back to previously written code. External documentation of source-code is supplemental to internal code commenting. As how the requirements documentation presents the software functionality, external code documentation presents an organized and detailed explanation of the source code. External documentation can be created through internal manual methods, or third-party tools such as JavaDoc [31] that can auto generate documentation based on a specific style of comment writing in the source code. Extra documentation such as this can also help team members other than the programmers understand the code, which could be especially useful for quality assurance or high level of technical support.

Once code is finished, testing confirms that the code works and adheres to the design and requirements. Testability of software is a determinant for software maintainability [11], so testing documentation is important to maintainability as every change will require a test and subsequent regression testing to verify the change did not cause other functions to stop working. Each test is outlined with a test case, or set of test cases, that are detailed enough for any tester to perform while also keeping track of what changes are being tested, or what additional changes need to be made based on testing. Test documentation, like code commenting and design and requirements documentation, should be detailed enough to provide the tester with all the necessary information, but

concise enough so that the actual tests are easy to read and perform. Bad test documentation increases maintenance turnaround time which will inevitably cause extra cost.

Along with the four document types listed, another document is important to maintenance, the change request, and is briefly mentioned here. Change requests are the documents that initiate software maintenance. Detailed change request documentation is critical to the lasting impact of deciphering and implementing the change [3]. These can come from external sources, such as the customer, or internal sources, such as support or development. All four types of change, including bugs and errors, are initiated by a change request. A good change request will state the issue, where it can be found, steps to create or recreate the issue, and, if possible, the solution. In the case of a problem, steps to recreate the problem ought to be simple enough that a tester can perform the test, but detailed enough that a programmer may be able to trace the issue in code. More detail can be added as the issue is tested.

Documentation is an important component of maintainability for a software system. Without documentation, there is no base from which change can begin, thus causing maintenance costs to rise. In order to prevent future costs, it should be beneficial to have a standard documentation process for those systems that have a focus for maintenance. A structure for setting up documentation for a more maintainable system is desirable and is proposed in the proceeding chapter, followed by a discussion in chapter IV.

CHAPTER III

METHOD

As the level of technology rises, so too does the freedom to create software that will meet the growing needs of clients and businesses. Increased technology comes with a price however, not only in hardware or software cost, but in maintenance efforts as well. Maintainability is concerned with handling this cost, be it from effort or financial means, by reducing the difficulty of making necessary change over time. The previous chapter outlined what to look for in order to make a more maintainable system, as well as present current known methods for analyzing the current level of maintainability in software. However, these methods are primarily architecture based and rely on maintenance scenarios for change. Scenarios are also highly suggestive and will differ in meaning from one developer to the next, whereas documentation should be consistent and concise for all within an organization.

Requirements present customer needs, architecture implements the requirements, code implements the architecture, and testing verifies requirements and code. The requirements document is recorded first in documentation-driven software development processes and the choices made and level of detail presented in them has a lasting effect on the rest of the development process. Poor requirements may lead to bad or incorrect design which likely leads to bad code. Transitively, bad or poorly written code and code

documentation is a result of unsatisfactory requirements. Thus, documentation is an important element to maintainability as the information presented allows for a higher level of understanding of the system's function, structure, and intent. Document-driven development processes are still widely used and hold roughly 1/3rd of developer interest [37]. The method being presented here is under the assumption that a document-driven process, such as the waterfall model, is being used. The following paragraphs detail a proposed solution to obtaining maintainability through documentation; first by showing how architectural maintenance tactics can be applied to documentation, as well as introducing three new tactics to the set, and finally, detailing how they can be applied to the different areas of documentation, followed by a case study at the end of chapter.

Software Documentation Tactics

The three tactics for achieving maintainability, as described previously, should translate well from architectural design to documentation, with slight differences between them. Cohesion among documentation and artifacts provides a greater sense of organization and, as with architecture, allows the documentation greater focus. Coupling within documentation, as with architecture, should be kept low as the higher the coupling among document artifacts grows, so too does the amount of change needed to complete edits to the documentation. Size for documentation can be a detriment, as the larger the document gets, the harder it may be to understand or find desired artifacts. However, this can be offset by a higher level of cohesion and by the amount of detail provided. The original three tactics are detailed below along with definitions for additional tactics: understandability, traceability, and networking.

The first tactic for document maintainability is cohesion, which is about keeping focus within the document and its artifacts. A high level of document cohesion means that details from other documents or artifacts do not spill over into one another. For example, a requirement for making withdrawals in an ATM system would only detail withdrawals, not deposits. Increased focus can be typically obtained by keeping a higher level of organization; in this way, cohesion is related to how well a document is organized.

Coupling is the second tactic for document maintainability and is about limiting repetition among documents. Requirements, component definitions, code comments, or tests that are repeated often, instead of referenced, increase the effort required to update the documentation as changes need to be made anywhere the content is repeated. For example, test cases that have a repeating sequence of steps included in multiple other tests should reference a master sequence instead of repeating the full steps. This process of referencing is part of the sixth tactic and is discussed at the end of this section.

The third tactic for document maintainability is size and is about keeping a desirable ratio between size and detail. A larger program becomes harder to maintain, and in the same way so too does documentation. In some cases, a large requirements or testing document is unavoidable, such as in larger and more complex programs requiring hundreds of requirements and perhaps even more test cases. It is here that a balance should be struck between how large documentation becomes with the level of detail provided. For example, a large document set with a high level of detail and understandability will be easier to understand and maintain than one with very little or hard to understand detail.

A fourth maintenance tactic for documentation is understandability. Understandability [11] is the measure in which how easily something, documentation in this case, is to be understood by the user. For documentation to be understandable, characteristics [11] such as conciseness, legibility, and self-descriptiveness need apply. Documentation that is written poorly or with ill-defined shorthand, and is contradictory will be harder to understand and consequently more difficult to update. For change to be implemented easily, the documentation should be understandable by all those that will be primarily involved in the different document levels, including the stakeholders, developers, testers, and clients. Understandability can also balance other tactics such as cohesion and size. Finally, where cohesion benefits from document organization, understandability benefits from organizing further into hierarchal categories, both within the documentation and its artifacts. This will be shown in the subsequent section.

Self-check-in

Allows guest self-check-in of a room, and its access.

Inputs:

Reservation Number – A unique number identifying a reservation. A reservation is a guarantee that a room will be held for the customer. This is required.

Name – Full name of customer. First and Last names are required. Middle is optional

Email Address – customer email. Must provide the email name and domain (person@somedomain.com for example). This is optional.

Credit Card – Stored electronic payment of the customer. Card number, name, and expiration are required.

A ResNum must already be assigned in order to proceed. CC entered should be verified to be a valid card per CC standards. Reservation information will be updated with entered data. Finished function will mark the reservation as Checked in and room key is printed as output

Figure 1: Poor Requirements Example

Before continuing with the final two maintenance tactics, an example of poor implementation of these four tactics is shown in figures 1 and 2 which represent a functional requirement entry in the Requirements document and its corresponding test case in the Testing document respectively for a fictional program. Since the program does not exist, nor is there a full document set, the figures do not represent any specific documentation standard, such as IEEE, and are here solely to represent the tactics provided in this method.

Self-check-in

Ensure that the Self-check-in module works.

1. Enter assigned Reservation Number
2. Enter name
    a. First
    b. Middle
    c. Last
3. Enter email address
4. Enter credit card
    a. name
    b. number
    c. expiration

A successful test will update the fields and change the reservation to checked-in.

Figure 2: Poor Testing Example

Cohesion is low in both documents as information is presented with no organization provided by section headers to let the reader know what exactly it is they are reading. The requirement document also details other functions or pieces of the program that do not need to be mentioned, such as what a reservation number or email address is. Consequently, this low cohesion provides high coupling as the definition for reservation

and email ought to be placed elsewhere in the requirements documentation, and a change in their definition will require a change here as well. For size, both documents are small but they provide very little detail as to what the objective is, or they provide too much individual detail causing the high coupling. Understandability is also low in both documents. In the requirement, there is an abbreviation for credit card, “CC”, that is not mentioned and there is a lack of clear objectives in the test document. Prompt visibility of whether or not inputs are required without reading through the full description is also absent from both documents.

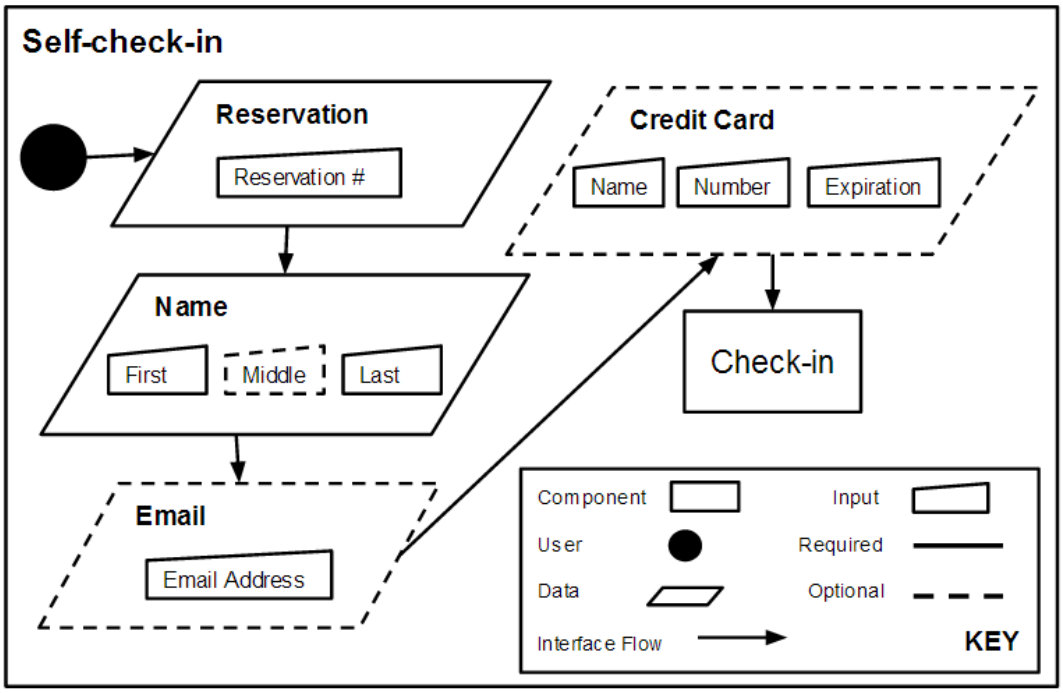

Figure 3: Design Example

Proper representations of the first four tactics are shown in Figure 3 and Figure 4 representing the design interface flow and source-code commenting respectively for the requirement in figure 1. Actual source-code is omitted and the commenting uses the

JavaDoc style while observing a desirable level of cohesion, coupling, size, and understandability. Comments do not go into detail of any function outside the scope of the class file, repetition is kept to a minimum, there are adequate descriptions to go along with each section of code, and the comments are written in a way that most programmers should understand. The interface flow design document shows the flow in which the interface for the self-check-in process takes. Solid lines represent required data, while dashed lines represent optional data.

The fifth maintainability tactic for documentation is the level of traceability within each document type. Traceability is the ability for documentation to be followed, or traced, through its history and application; and increases maintainability by providing a better understanding of past or future changes to many different people involved in the software's development [18] [27]. Traceability is used in a variety of concepts and tools such as software configuration management [15], a process for tracking changes in code versions; and third party change request, bug, and test case repositories such as FogBugz [17] which provide a means for organizing and tracking testing documentation. During the life-cycle of a product, the software will undergo many different modifications, typically by many different people, which oftentimes go undocumented causing degradation of information and increased maintenance workload [20]. A documented history of change within each level of documentation should provide both higher understandability as well as future ease of maintenance. Figures 1, 2, 3, and 4 are absent of any traceability, and will be added in the case study.

Building on the idea of traceability, the final addition to the maintenance tactics set for the different levels of documentation is a concept of internal and external

```
1  /**
2   * The S_Check-in class is used to check a customer into their
3   *  assigned room and print room keys.
4   * <br>  keyboard: An instance of Scanner for user input
5   * <br><br>
6   * @author John Doe
7   * @version 1.0
8   * @since 3/25/14
9   */

11  public class S_Check-in {

13  /**
14   * main calls the methods that take user input from the user.<br><br>
15   * main stores values entered from the user and uses those
16   *   values to update the reservation through the UpdateRes.
17   * Once information is updated the room will be
18   *  marked as checked in through the CheckIn method.<br>
19   * <br>  name: [Object] object of class Name (required)
20   * <br>  rNum: [Long] guest's reservation number (required)
21   * <br>  email: [String] guest's email address
22   * <br>  card: [Object] object of class CreditCard
23   */
24   public static void main(String[] args) {
25      .
26      .
27      // Call getName Method and assign to name
28      .
29      .
20      // Call getCard method and assign to card
30      .
31      // Call UpdateRes
32      // Call CheckIn

34   } // end main

36   /**
37    * getCard creates a CreditCard object and assigns it to
38    *   card in main.<br><br>
39    * getCard asks if a credit card will be used. If yes,
40    *   ask guest to enter a card name, number, and expiration.<br>
41    * Credit card information will be verified by ValidateCard.<br>
42    * Fields may not be left blank.<br><br>
43    * @param card-in is the blank card object created in main.
44    * @return sends back a CreditCard object with related info.
45    */
46    public static CreditCard getCard(CreditCard card-in){

48    } // end CreditCard

50  } // end S_Check-in
```

Figure 4: Source-code Example

document linking, hereby called document networking, or simply, networking. Much like networking among people, networking is concerned with providing references between all the different levels of documentation, rather than simply providing history tracking. What this provides is a quick reference lookup between documents both internally and externally. In this way, there should be less time spent going from requirements checking, to design study, to code implementation, to test case generation or editing; as well as less time spent attempting to understand all assets of a particular requirement, design, piece of code, or test case. Internal networking requires that each artifact, different requirements within the requirements document for example, must be referenced to one another within each requirement listing if they are in some way related. External networking requires that each item of documentation references its corresponding document within one of the other three documentation levels. External networking creates a link between the requirements, design, code, and testing documents.

A typical change request starts with the request for change, followed by a validation of the requirements and design to determine if the change is warranted, then by either a test if the request is for a bug, or a change in code, followed by a repeated cycle of testing and code changes until the request has been fulfilled and closed [45]. Each time during the maintenance and testing phase when a change is made that requires a change in either of the other documents; the change should be made there as well, as seen in figure 5. This provides consistency among all documents which lowers the amount of document degradation over time, increasing ease of maintenance. Document networking increases the ease in which this process takes place by allowing quick reference points for each document type and their individual artifacts and provides consistency when change

occurs. Figures 1, 2, 3, and 4 are absent of networking, and will be added in the case study.

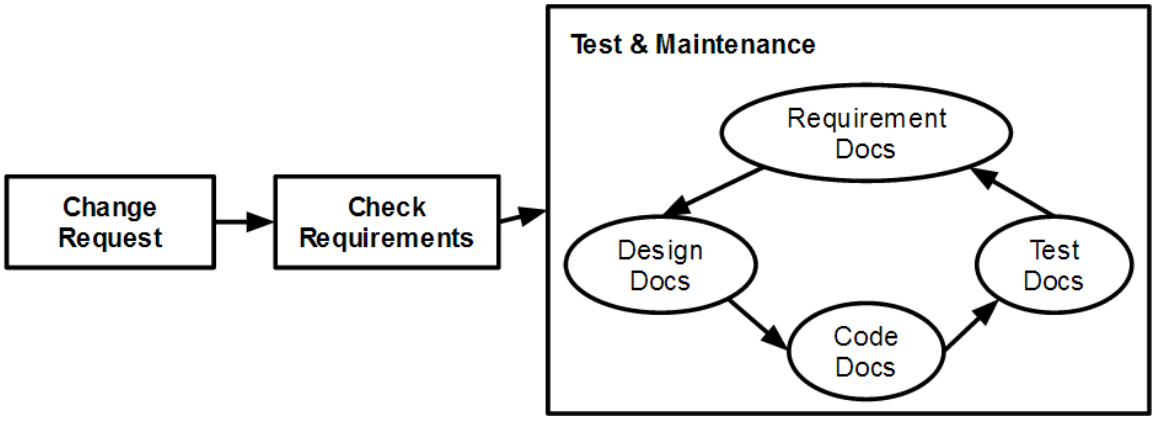

Figure 5: Documentation Cycle

Both coupling and cohesion, while not specifically stated as such, are used in current known standards for requirements and design documentation. IEEE 830 [26] states in section 4.3.7, that the requirements document must be modifiable. By modifiable the guideline suggests the document should not define or detail the same requirement more than once (coupling) and should be organized with a table of contents and index (cohesion). Tracing requirements throughout the document, section 4.3.8, is also listed as a guideline requirement and states that each requirement be made traceable backward and forward throughout the requirements document. Backward and forward tracing works like a tree structure where backward traces are the parents of the current document artifact, and forward traces are the children of the current artifact. Networking is derived from this and is taken a step further by adding that all four document types be referenced backward and forward externally and internally as described in the previous section.

Application of Software Documentation Tactics

This section will describe the proposed guidelines for achieving a higher level of maintainability. There are six in total, each applying the different maintainability tactics from the previous section:

1. Cohesion (increase focus),
2. Coupling (lower repetition),
3. Size (size and detail ratio),
4. Understandability (language and presentation),
5. Traceability (change history), and
6. Networking (referencing and consistency).

The following paragraphs will detail these six tactics and describe how they can be applied to the requirements, design, source-code, and testing levels of documentation either during or after development. An example of their application is presented in the case study following this section.

First is the application of cohesion. Cohesion in the requirements document and SDD is about organization and keeping to the adage "a place for everything and everything in its place". Both documents as a whole should fit within designated sections each with different goals, such as functions, interfaces, components, third party tools, operating environments, business practices, glossary, constraints, etc.; with each artifact organized in a similar manner. Artifacts will adhere to the same rules as low level architecture cohesion in that they only entail information that is relevant to them. Function processes, for example, should not detail constraints and vice versa; a glossary or index of terms used within the entirety of the requirements document should not

include anything other than definitions or locations where the terms were used; and so on. These principles apply to source-code and testing documentation as well. Individual code classes, lines, or functions need only be relevant to that specific item which is being commented. Test cases within the testing document need only detail their specific case. External documentation for code and an overall testing document, if one exists, ought to as well be organized in a categorical manner.

Second is coupling and is concerned with ties between requirements, design, testing, and code that will require parallel change with another entry in that document category. Links such as these cause repetition, and the more repetition there is between different parts of each artifact or different artifacts, the more work involved in changing the document as needed. A situation where changing the documentation of one article will require the same change in another is not desired. For example, if a function in code is commented on, those comments should not be attached to that same function when used in another piece of code; rather it should be referenced. As another example, a requirement definition made in the requirements document should not be made again in a separate article within the same document or in the subsequent test case. Some coupling is inevitable, such as coupling of a requirement definition to its test-case and source-code definitions. In cases such as these it may be more beneficial to keep the same description so that when one changes, it can be easily copied to another.

The third application is size, which can be misleading in regard to documentation. In evaluating architecture for size, the concern is about keeping components and code short since the larger the size and complexity, the lower the ease of maintenance. For documentation, the detail provided along with size is what is important. Vague or missing

information lowers the ease of understanding and making required changes. However, a large requirements document, for example, with a high degree of detail and cohesion and low coupling should balance out the amount of work needed to update a larger document. Small test cases can become harder to perform if the test description and steps are vague. Comments in code are largely dependent on how well the code is written. Poorly written code will most likely yield the need for more comments, but if code is written in a well understood manner, less commenting may be needed. External code documentation with higher levels of detail is also desired and can balance out bad or low levels of commenting. Architecture design documentation should be very detailed as to how the design and relationships among components is defined.

Guideline four is ensuring understandability. Documentation is only as good as the way it is presented for whom it was created. Internal and external code documentation and low-level design diagrams can be more technical as it is meant primarily for programmers; however the level of technical detail should consider the level of expertise of the programmers who will be using it. The requirements document may need to be split into different versions, one for stakeholders, another for users (a user manual), and another for the development team; with different sections in each providing separate topics such as hardware, software, and implementation [22]. A user manual version of the requirements can also be beneficial for entry level customer support. Finally, testing documentation should be understandable not only by programmers doing initial unit testing, but by the QA and testing teams if applicable. More technical details for initial unit testing can be included at the beginning of a test case, but as it gets passed into use testing, the detail and understandability should be brought down to a simpler state.

Guideline five is the inclusion of change history for the assurance of traceability. Dates when changes were made, names of the people that suggested or implemented the change, version changes, and what was changed are all items to consider adding to documentation. This could be done in a number of possible ways [27] [41]. The addition of a history log within requirements, design, external code, or testing documentation that tracks and lists these items in an organized manner such as by date or person is one such way of including traceability. Artifacts could also include their own history. Source-code commenting, for example, could comment where code was updated, or at the top of the class with a list of changes as they have occurred. A functional requirement within the requirements documentation may have a history section which describes when the requirement may have changed. Testing documents especially will want traceable information as not every test passes the first time it is executed and the same person may or may not be working on the test as it plays out, so a detailed history is required.

The final guideline is the networking among documents and artifacts. Requirements and design decisions that are realized in code should be linked, and test cases created for those requirements should be linked as well. Artifacts should also be linked, both backward and forward. The goal is to ensure that each document type has a way to easily find a corresponding entry in another, or its own, document. A simple way to realize networking between documents is to establish a library referencing system where each item (line, function, or class code; entry in requirements; design diagram, or test case for example) has an ID and that ID is used to reference itself in other documents. This ID should always be accompanied by the corresponding ID from the other three document levels. If no corresponding entry in another document is available then it may

be omitted, however this should be rare as any entry in the requirements document should be realized in design and code, and that code should have a means to be verified against the requirements. Networking such as this can increase user understandability of a software feature and consistency of documentation as a whole by providing reference points for additional information from other documents, whether internal or external and is already used in established tools such as FogBugz. When setting up the documentation for networking it is important to look at how each document is linked to one another and does the information provided in each document allow for easy location of a corresponding entry in another document.

Through the use of varying evaluation methods, the application of maintenance tactics on software architecture has been shown to be effective in increasing software maintainability. Similar application into documentation should provide similar results. Standard architecture maintenance tactics such as cohesion, coupling, and size allow for less work when change is needed in multiple documents. The addition of understandability, traceability, and networking focus on lowering the time needed to understand and locate different components of documentation in order to validate and implement maintenance requests by providing history and references to related items. Using an ID system may also lend itself well to an automated documentation system which could further lower the turnaround time of different aspects of the maintenance process, and is discussed in chapter IV. While this method has not been tested or applied to a live project, a case study demonstrating the concept is presented in the next section.

Case Study

Due to the large scale nature of examining a system's documentation in full, a light demonstration of the method will be presented here. Depending on the size of a development project, requirements, design, source code, and testing documentation, could take up four separate case studies alone. As this method is only a proposal, a large scale evaluation is left as future work; in lieu of this, a demonstration of the application steps from the previous section is presented here. Figures 1, 2, 3, and 4 will be used as the base and will be progressively improved using steps 1-4 and finished with steps 5 and 6 while solving a simple change request.

The sample documentation set describes a simple self-check-in form for guests to use at a motel so that they can quickly obtain a room key and head to their room when an attendant is either busy with other guests or not available. As mentioned prior, this is purely a hypothetical example and not part of a system currently on the market or in development. The module will allow entry of the guest's reservation number; first, middle, and last name; email address; and credit card. The reservation number, first name, and last name are required fields per the requirements in figure 1. For simplicity sake, the source-code is omitted and only some of the internal code documentation was provided in figure 3. Code implementation is in Java and the documentation presented will be using the JavaDoc style. Not all comments will be shown, and breaks in the example are determined by a period (.). Design documentation is represented by an interface flow diagram and was shown in figure 4. Test case documentation was presented in figure 2. The test case is considered a master case, or unit test, where the steps described are what are needed to test the module when all the requirements are

followed. Many sub-tests would also be created for exhaustive testing. The issues with these documents in regards to cohesion, coupling, size, and understandability were discussed earlier in the chapter and are not repeated here. The following paragraphs will be applying guidelines 1-4 to figures 1 and 2 in two stages: first cohesion and coupling, followed by size and understandability.

Applying cohesion to the requirement and test case artifacts requires the sectioning off of information into defined parts. Headers for description, inputs, outputs, preconditions, and post conditions are added to the requirement, while description, test steps, and desired outcomes are added to the test case. This allows for the information to be grouped and separated while also becoming more understandable. Coupling application lessens the repetition level of the document. The prior inclusion of input and output definitions is removed in order to keep coupling low. The changes made to figures 1 and 2 can be found in figures 6 and 7 respectively.

While both documents are already fairly small, the information provided is light on detail and is not presented in an understandable way. Lists are not shown in a familiar list format such as numbering or bulleting. Shorthand is also used in pre and post requisites for the reservation number and credit card. Since these abbreviations are not defined, they should be removed. The description of the requirement and test case is updated to better reflect its goal, while the input and output lists are put in a numbered format. Inputs are now marked with a "*" in order to denote that they are required, rather than using the word "required" repeatedly. These changes allow for the documentation to be followed easier and increases visibility of information, such as the required inputs and desired outputs. Changes made to figures 5 and 6 can be found in figures 8 and 9

respectively.

Self-check-in

Description

Allows guest self-check-in of a room, and its access.

Inputs

Reservation Number –This is required.

Name – First and Last names are required. Middle is optional

Email Address – This is optional.

Credit Card –Card number, name, and expiration are required.

Outputs

Finished function will mark the reservation as Checked in and room key is printed.

Preconditions

A ResNum must already be assigned in order to proceed.

Post conditions

CC entered should be verified to be a valid card per CC standards and reservation information will be updated with entered data.

Figure 6: Cohesion & Coupling - Requirement

Self-check-in

Description

Ensure that the Self-check-in module works.

Test Steps

1. Enter assigned Reservation Number
2. Enter name
    a. First
    b. Middle
    c. Last
3. Enter email address
4. Enter credit card
    a. name
    b. number
    c. expiration

Desired Outcomes

A successful test will update the fields and change the reservation to checked-in.

Figure 7: Cohesion & Coupling - Test

Self-check-in

Description

Allow for a customer to check themselves into an assigned room when an attendant is unavailable.

Inputs

The following is a list of inputs used. Required fields, as well as their sub-parts are marked as such with a * symbol:

1. Reservation Number *
2. Name *
    a. First *
    b. Middle initial
    c. Last *
3. Email Address
4. Credit Card
    a. Name *
    b. Number *
    c. Expiration *

Outputs

1. Reservation marked as Checked in.
2. Room key is printed.

Preconditions

1. A Reservation number must already be assigned in order to proceed.

Post conditions

1. Credit card entered is verified valid per credit card standards.
2. Reservation information updated with entered data.

Figure 8: Size & Understandability - Requirement

Traceability and networking will now be added alongside a change request for the requirement in order to show their importance. The problem with the requirement listing in figure 8 is that the credit card itself is not required, only the credit card name, number, and expiration are shown as required. This leads to an incorrect implementation in design and code. The “Name” field is marked required along with its sub-components, first and last, so while designing the credit card portion of this module, it was falsely understood to mean that the credit card was optional, but once one was used, the sub-components

Self-check-in

Description

This master case is used to ensure that the Self-check-in module works as described by its requirement: to allow a customer to check themselves into an assigned room when an attendant is unavailable.

Test Steps

1. Enter assigned Reservation Number
2. Enter name
    a. First name
    b. Middle initial if desired
    c. Last name
3. Enter email address if desired
4. Enter credit card if desired
    a. Card holder's name
    b. Card number
    c. Card expiration

Desired Outcomes

1. A successful test will update the appropriate fields within the reservation as described in the requirements; and mark the reservation as checked in.
2. At no point should the operation crash, or present an error that cannot be rectified.

Figure 9: Size & Understandability - Test

were not. This mistake is then translated into the test case(s).

Consider the case where a change request is made to solve this issue, by stating that the credit card input is supposed to be required and cannot be left blank. Making the change in requirements would be relatively painless, as all that would need to be changed is adding a required symbol next to input 4 in figure 8. The design, source-code and test documentation will follow the same type of change. Source-code documentation will need to update the credit card to "(required)" in line 22 of figure 3. Design documentation will make the "credit card" portion a solid line instead of a dashed one. Test case documentation will remove the "if desired" line from step 4 in figure 9. Each update to

the documents individually presents no concerning issue and the time taken to make each update should be fairly quick for even a relatively slow individual or someone new to the process. However, if these documents are part of a much larger program with hundreds or thousands of other requirements, design diagrams, code entries, and test cases, the layout presented could cause an unnecessary time sink in finding each entry to make the change. This is where the linking, or networking, of the documents is desired. Each document and how networking can be applied, using an ID system as described from the previous section, is presented next.

For this example, ID tags will start with a letter designating what document the item belongs to, followed by a number which uniquely identifies that item within its respective document. The requirements document will use ‘R’, design will use ‘D’, source-code will use ‘SC’, and testing will use ‘T’. It may be desirable, in larger systems that require longer documentation, to also include sub-IDs that designate which section of a document the item can be found; for instance, use-cases within the Requirements documentation may have a ‘U’ tag along with the ‘R’ to form something like ‘R-U-12’ to indicate it is the twelfth use-case within the Requirements document, or an ‘F’ in code documentation to denote a function within a class.

First is the requirements document where “Requirement ID” is listed immediately following the title, and once again in the body under the “Requirement” label. Listing the ID in the title offers direct visibility of the ID in a header style, while listing it again in the body reaffirms the ID if the reader examines the requirement further. In the event use-cases are used within the Requirements a use-case ID is listed so that the reader can quickly find that artifact and view a more detailed process on the how the requirement is

used. Next are the related and coupled requirements. Related requirements allows the listing of those requirements that either this requirement uses or is used by which provides a higher link level between document entries. Coupled requirements are those requirements that are directly affected by the changing of this requirement and may require a change as well. When a change is needed for this requirement, the designer or programmer knows what requirements may also need changing. This also has the added benefit of showing how coupled the system may be between functions or modules in the eventual or current design and architecture. Design documentation such as the interface and class diagrams is referenced next. The source-code and test case tags provide the reader with the location of the matching source-code and test case. For the rest of the document, different artifacts that may have an entry elsewhere within the document are also listed. The listings within the inputs and outputs sections are labeled with their corresponding entries in a glossary of terms within the requirements document for more detailed definitions; the requirements listed under "post condition" have a link to their respective articles as well. These extra ID listings do not list corresponding entries in source-code or testing to keep cohesion within the article high. The reader can go to the desired article to get the required source-code or test case IDs for other requirements listed. A "Change History" section has been added for traceability. In this section, the dates, name(s), and descriptions of changes are recorded. The final version of the requirement listing is presented in figure 10.

Design documentation with traceability adds a "Change History" section for changes made to this design artifact. Networking application adds a design ID to the title and references to the other three document types are listed together underneath it. If more

Self-check-in (R-124)

Requirement:	**R-124**
Use Case:	**R-U-30**
Related Requirements:	**R-199, R-50, R-75, R-187**
Coupled Requirements:	**R-123**
Design Interface:	**D-56**
Design Class:	**D-197**
Source-code:	**SC-200**
Test Case:	**T-176**

Description

Allow for a customer to check themselves into an assigned room when an attendant is unavailable.

Inputs

The following is a list of inputs used. Required fields, as well as their sub-parts are marked as such with a * symbol:

1. Reservation Number * (R-700)
2. Name * (R-701)
    a. First * (R-701a)
    b. Middle initial (R-701b)
    c. Last * (R-701c)
3. Email Address (R-702)
4. Credit Card * (R-730)
    a. Name * (R-730a)
    b. Number * (R-730b)
    c. Expiration * (R-730c)

Outputs

1. Reservation marked as Checked in. (R-75)
2. Room key is printed. (R-187)

Precondition

1. A Reservation number must already be assigned in order to proceed.

Post condition

1. Credit card entered is verified valid per credit card standards. (R-199)
2. Reservation information updated with entered data. (R-50)

Change History

1. 4/15/14 : Jane Doe : Updated (R-124) and (SC-200) to force entry of Input 4.

Figure 10: Traceability & Networking - Requirement

design documents need to be referenced they could be added here as well. Each data module within the diagram is also listed with their corresponding definition in the requirements document. The “Check-in” process show’s its source-code link and could also include the requirement and design if desired. Other design documents such as the UML class diagram could add each of the source-code IDs or the component diagram could link to other architectural design diagrams or requirement artifacts. The final version of the interface design flow is presented in figure 11.

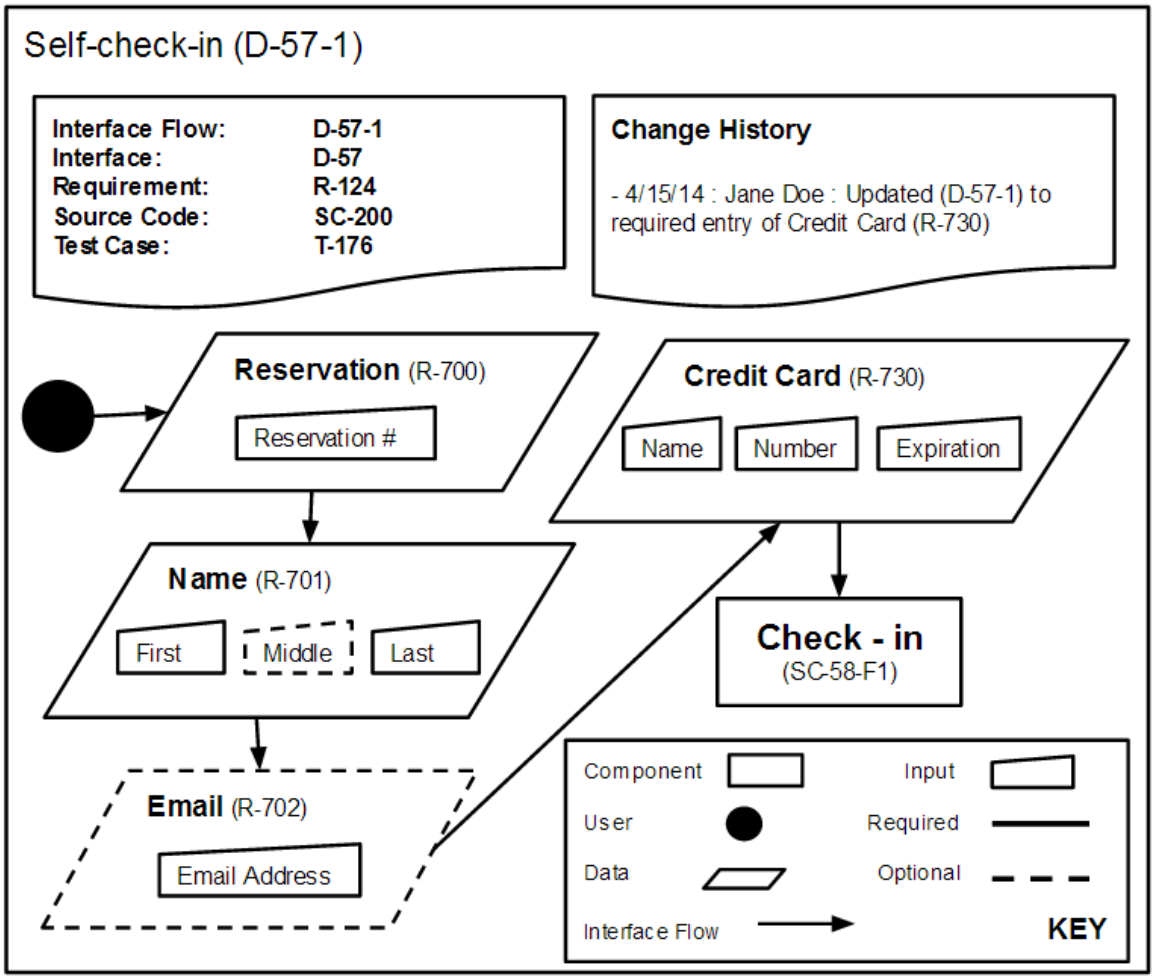

Figure 11: Traceability & Networking - Design

Test case documentation with traceability and networking sees a similar change as the Requirements, but on a smaller scale. The title ID tag remains, while the initial "Requirement" field is moved to the bottom of the ID list and in its place is the test case ID. The use-case and coupled listings are removed, while the related field remain the same. Related test cases are those that are sub-cases of the current test case, such as those used for exhaustive [14], or path and bound [28] testing of the test-case, or those test-cases that are used in the completion of the current test case. The "x" at the end of the related test cases is used in lieu of listed all the cases. As with the Requirements, the source-code ID(s) that belong to the test case are listed as well as the design interface ID. Finally, a "Test History" section has been added for traceability; note that details about actual test results are not provided as they would be in the test case testing notes which are not provided. The final version of the test-case listing is presented in figure 12.

Changes work slightly different in the Source-code documentation, but for the most part provide the same style of traceability and networking between the other comments in code and the other types of documentation. The main ID of the class should be displayed at the beginning, as seen in line 6, followed by the corresponding requirement(s), design artifact(s) and test case(s), shown in lines 7, 8, and 9. Whenever an update is performed, the version and date of the code, if listed, should be updated along with the name of who made the change and the ID of the field or function that was changed. Next, the main method for the class will include an ID, denoted by an M in line 20, and each other method used but not belonging to this class should include their respective IDs for easy lookup. Individual functions within the class will have their own ID, denoted with an 'F' as shown on line 45, and if a test case is available for that

Self-check-in (T-176)

Test Case: T-176
Related Test Cases: T-176-1, T-176-2, T-176-3, …, T-176-x
Requirement: R-124
Design Interface D-56
Source-code: SC-200

Description

Ensure that the Self-check-in module works as described by its requirements entry.

Test Steps

1. Enter assigned Reservation Number
2. Enter name
    a. First name
    b. Middle initial if desired
    c. Last name
3. Enter email address if desired
4. Enter credit card
    a. Card name
    b. Card number
    c. Card expiration

Desired Outcomes

1. A successful test will update the appropriate fields within the reservation as described in the requirements; and mark the reservation as checked in.
2. At no point should the operation crash, or present an error that cannot be rectified.

Test History

1. 4/15/14 : Jane Doe : Initial unit test of (T-176) completed. Requires use testing.

Figure 12: Traceability & Networking - Test

particular function it should be listed as well. As with the main method, any other functions being called by this function should have some comment listing the specific ID. As with the requirements and test case documentation, this linking of comments does not indicate actual code coupling, rather it provides a higher level of networking among code. The final version of the source-code is presented in figure 13.

```
1  /**
2   * The S_Check-in class is used to check a customer into their
3   *   assigned room and print room keys.
4   * <br>  keyboard: An instance of Scanner for user input
5   * <br><br>
6   * Source Code: SC-200 <br>
7   * Requirement: R-124 <br>
8   * Design: D-56, D-197
9   * Test Case: T-176 <br>
10  * Updated 4/15/14 by Jane Doe: SC-200-M, SC-200-F4 <br>
11  * @author John Doe
12  * @version 1.01
13  * @since 4/15/14
14  */

16  public class S_Check-in {

18  /**
19   * main calls the methods that take user input from the user.<br><br>
20   * Main: SC-200-M<br>
21   * main stores values entered from the user and uses those values to
22   *   update the reservation through UpdateRes (SC-57-F1).<br>
23   * Once information is updated the room will be
24   *   marked as checked in through CheckIn (SC-58-F1).<br>
25   * Room key will be printed through PrintRoomKey (SC-60-F2).
26   * <br>  name: [Object] object of class Name (required)
27   * <br>  rNum: [Long] guest's reservation number (required)
28   * <br>  email: [String] guest's email address
29   * <br>  card: [Object] object of class CreditCard (required)
30   * @param args is required for main, but not used by this method.
31   */
32   public static void main(String[] args) {
33      .
34      .
35      // Call getCard method and assign to card
36      .
37      // Call UpdateRes
38      // Call CheckIn
39      // Call PrintRoomKey

41   } // end main

43   /**
44    * getCard creates a CreditCard object and assigns it t card in main.
45    * <br><br>Method: SC-200-F4
46    * <br>Test Case: T-176-35
47    * getCard asks guest to enter a card name, number, and expiration.
48    * Credit card information verified by ValidateCard (SC-206-F1)
49    * Fields may not be left blank.<br><br>
50    * @param card-in is the blank card object created in main.
51    * @return sends back a CreditCard object with related info.
52    */
53    public static CreditCard getCard(CreditCard card-in){

55    } // end CreditCard

57  } // end S_Check-in
```

Figure 13: Traceability & Networking - Source-code

The example shown here demonstrates how the addition of documentation tactics for maintainability add a higher level of organization, understanding, and reference among the different documents involved in the development, testing, and maintenance of software. Much like a relational database, systems such as ID tagging allow for greater ease in finding related documents when making change, as well as ensuring consistency. While a small system may or may not benefit from this, it is not difficult to see how in a large system this is an added benefit for incorporating change among all affecting documents where there are potentially hundreds of requirements, design diagrams and artifacts, code files, and test cases to search through. An added benefit of labeling code is the referencing of different code fragments that may be harder to find when the software is large and the number of code files is great. The ID system employed here is only an example, and there could be other ways of achieving similar goals that better suit the needs of the business and developers involved.

Turnaround time of a maintenance request involves understanding the request and determining if the change should be carried through, then once an understanding has been made the implementation needs to be understood and performed, and finally the related documentation needs to be updated so that information remains consistent and does not become degraded. The faster this process completes, the sooner work can be done on other areas of development which cuts time cost; and applying networking, as well as the other five tactics, among the different levels of documentation should provide this benefit. The final chapter provides possible future application of this method through evaluation and automation techniques, followed by closing comments and final thoughts.

CHAPTER IV

DISCUSSION

Future Application of Method

The application example shown in the prior section shows a literal translation of the method proposed in chapter III. This method however is not limited to one application for maintainability and, given further in-depth research, could be used as a base for other ventures in increased maintenance interests. Some possible extensions, in the area of evaluation and automation, of the method are discussed in more detail in this section.

Maintenance tactics and scenario generation for architecture have been used to create methods for evaluating the degree of software maintainability, including those listed in chapter II, among others [8], [21], [2], and [13]. As with these, translating the method proposed in this paper should be a relatively straightforward process. An evaluation method for judging software maintainability through documentation would use the six defined tactics: cohesion, coupling, size, understandability, traceability, and networking, as metrics for maintenance suitability analysis rather than applying them directly to the documentation. One example of an evaluation model could apply the qualitative approach used by Shaw and Garlan [43]. Each tactic could be used as a point of evaluation against the four levels of documentation: requirements, design,

source-code, and testing; each receiving a rating given by the evaluator(s). The rating scale for results can be presented in varying ways, depending on the needs of the evaluation; examples include: numbered or alphabet grades, symbols such as + and -, quality words such as good or bad, low or high, etc. An example of this, unrelated to the example in the previous chapter, is provided in figure 14. A quantitative method [11] could also be derived; however well-defined metrics for all 6 tactics would need to be generated in order to give an accurate returned value for maintenance cost or difficulty prediction.

	TACTICS							
			SIZE					
	Cohesion	Coupling	Size	Detail	External	Understandability	Traceability	Networking
Requirements	+	++	-	++		+/-	-	++
Design	+/-	+	+/-	+		++	-	+
Code	+	+	+	-	N/A	+/-	-	+/-
Testing	+	++	+	-		+	--	-

Figure 14: Evaluation Example

The method provided in chapter III and its subsequent case study show how, once set up, the time required to understand and complete a maintenance request can be shortened through documentation maintainability tactics. However, this method is one that is done manually and would still take time to set up and provide upkeep of all four documentation levels. Networking and traceability among physical documents, not stored electronically or online, provides quicker reference to documents but the lookup must still be done manually by those in need of them and ensuring consistency among change is not guaranteed. Studies [32] have shown that manual documentation processes such as

this have been ineffective or unused due to a lack of interest among software engineers in both the creation and upkeep of requirements, and especially, source-code documentation. Without multiple extended case studies on both small and large systems, the benefits of the method on future maintenance could be outweighed by the increased effort in setup and update time. Automated tools for referencing or updating documentation would alleviate this manual effort, and many are available or have been researched for traceability [18]. Techniques such as automatically acquiring traceability links between requirements and code [34], automated retrieval of links found in requirements [23], and automated traceability updates through design documentation such as UML [33], are a sample of research done in acquiring proper maintainability of documentation to lessen the burden of the manual documentation upkeep process. Thoughts for expanding this research to networking are presented below.

The common concept between current automated techniques is recognizing tags for traceability so that different documents can be automatically located or updated. The notion of networking presented in this paper provides a similar start to possible research into an automated document networking system. The following are suggested automated applications of the method presented in this paper, through inactive and active means.

Networking forces each document to be linked to either themselves or others by reference IDs. Through semi-automated means, an inactive database of these IDs could be stored and used to query document information as needed. Information such as each requirement, design, source-code, and testing ID; file’s names and locations; ID descriptions; related or coupled documents; document keywords; and more could be stored in a single database table or, preferably, multiple database tables to provide

normalization. Once data is saved, finding a document by ID or keyword is as simple as creating a query template and updating appropriate values for each lookup. This provides a significant boost to effort required to find documents as the speed of a database query is far faster than what a human is capable of in comparable time. To illustrate this, a simple database table was created with 5000 entries. Columns in the table represent requirement, source-code, and testing document IDs along with their names and an overall description. Multiple queries were run to locate anywhere from 1 to 10 different requirement, source-code, and testing IDs within the 5000 data set. Query executions took anywhere from .004 to .06 seconds from a remote access location. In comparison, simply lifting up a piece of paper takes 1 second for a human, an at least 15 times increase in the time it takes the database to query 5000 lines of data and organize it as desired. Another benefit of querying is that results can be ordered and filtered in whichever way is desired by the user; for example, the results could display all coupled requirement IDs ordered by their source-code document locations so that the user can see if there is a cluster of source-code that would need to be changed.

Taking this idea further, the database could be used as a reference source for electronically stored documentation. Stored electronically and presented through a web page or some other means, such as a wiki, these references, made into hyperlinks, provide an instant look up to related documents, which lessens the time needed to understand a problem, by removing the need for user querying. Suppose a complex maintenance request comes in that involves many different requirements to change. With all documents linked, less time is spent searching for information, understandability of the problem is increased, and maintenance effort is lessened.

The prior automated suggestions still leave the issue of manually updating documentation. A possible automated solution is for specific sections for each document, such as a description, could be added to each artifact under the same tag so that when updated in one, it is processed and updated in all documents through a search algorithm. The database data could be used to store these similar references, such as the descriptions or any other common text between all four documents. If an ID, filename, location, etc. needed to be updated it would simply need to be updated in one place and all of references would update with it.

Active database solutions are also viable and could be provided through event-driven, or publisher, architecture [35]. Blackboard architecture [38] is one such possible direction to take for an architecture built around maintaining documentation and providing automated maintenance in both the software and documentation. The blackboard would monitor when a requirement, design, code, or testing artifact is changed and remind the developer that the corresponding artifacts need to be updated as well. These notifications would provide links to quickly access these effected artifacts. An active database that monitors state changes could call 3rd party tools or software whenever the state of the database is changed and take an appropriate action. Take the self-check-in case study used in the previous chapter as an example. Once the change request is found to be valid, the requirement entry could be updated which would trigger an event that allows other tools or components to update the rest of the documentation. Taking this further, if the blackboard, or a similar event-driven architecture, were to be built around fully automated maintenance; the event triggers could automate the entire change process along with the documentation. An example of this application is shown in

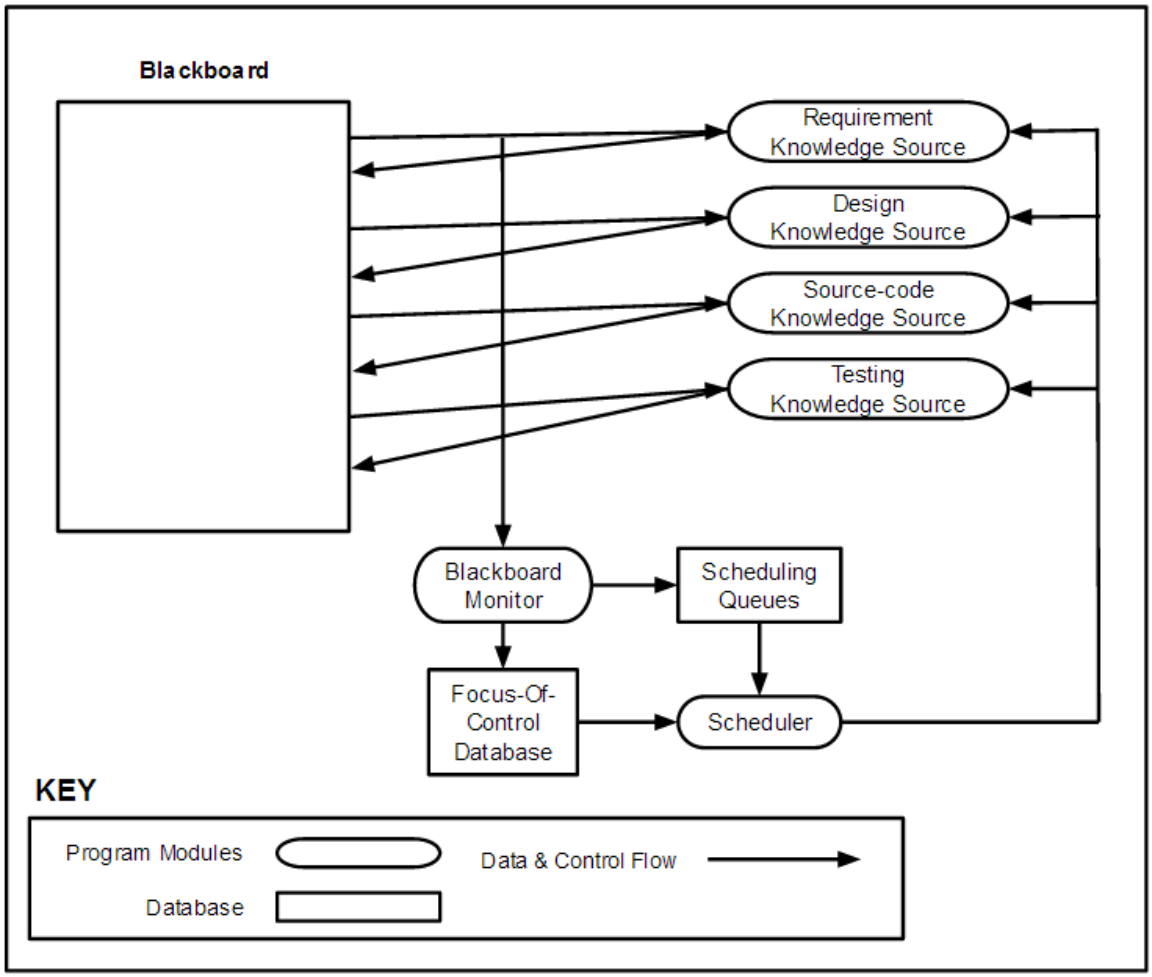

Figure 15: Blackboard Solution Example

figure 15 which was derived from a diagram of the Hearsay-II project found in [24]. As changes come into the blackboard, information is passed to the documentation knowledge sources through the scheduler and as more information is acquired about the problem, the different knowledge sources would provide updates to not only the documentation, but carry out the change as well. Consider the case where a function in code was changed in order to fix a reported bug, as an example. The developer would update the code, and the blackboard would detect this state change, send a request to the testing knowledge source to request a unit test be completed. Once the unit test is automatically carried out, if it is

determined a success, the information is passed to the requirements and design knowledge sources to provide updates to corresponding documentation that may have been effected by the fix. This application of blackboard, or other event-driven architectures, could provide a sufficient time, and subsequently financial, cost saving solution to the manual networking process detailed in this paper by providing automated software maintenance and documentation consistency. Further research on this and other, similar, solutions is left as open research.

Conclusion

Maintainability tactics and evaluation through architecture have been shown to be proven methods for determining maintenance suitability in software systems. However, software development does not begin with the architecture. Documentation plays an important role in software maintainability from the beginning of the development process with requirements definition, down to the continuing cycle of testing and maintenance. Poorly written requirements documentation can lead to misunderstandings of what is desired, leading to incorrect or poor architectural design which leads to poorly written or broken code. This will ultimately end in low maintainability causing higher costs on an already costly area of software development.

A proposal for decreasing maintainability effort through documentation, using low level architecture maintenance tactics, along with understandability, traceability, and networking has been presented. Using proven maintenance tactics, from software architecture, on documentation provides a set of general guidelines for achieving a higher degree of maintenance and understanding among the different documents involved in the

development process. These tactics offer a direct correlation with change and the ease in which change can be implemented. Understandability increases the ability to understand each document by those that require them. Traceability adds another layer of ease when trying to understand current and past versions of software. Finally, networking provides a reference point between all documents for quick acquisition of knowledge to understand the maintenance request and update effected documents during and after maintenance completion. The importance of external networking is in the forced consistency with updating all documentation and application to possible automated maintenance. Networking, both internal and external, along with grouping together these tactics for a set of guidelines is the main contribution of this work, as shown by figure 16. This figure summarizes the related work referenced in this paper toward the formation of the guidelines. The first column is the source as referenced in this paper, with this paper at the top of the list for easy reference. The next six columns are the six guidelines presented in this paper: cohesion (COH), coupling (COU), size (SIZ), understandability (UND), traceability (TRA), and networking (NET) in both the internal (INT) and external (EXT) sense. The next two columns represent if the source provided a qualitative or quantitative evaluation of maintainability, followed by the last two columns representing if the source was for documentation or architecture.

Maintainability is about measuring the ease and cost of change. By applying modifiability tactics, increasing understandability, ensuring traceability, and applying networking through the concepts and method provided in this paper, it can be observed that the ease of maintenance can be potentially increased which can bring down time cost, even more so by automated application. However, what of financial cost or the time

						NET		Evaluation		Type	
Source	COH	COU	SIZ	UND	TRA	INT	EXT	Qualitative	Quantitative	DOC	ARC
This Paper	x	x	x	x	x	x	x	x		x	
[2]	x	x	x						x		x
[6]	x	x	x					x			x
[8]								x			x
[9]	x	x	x						x		x
[10]								x			x
[11]				x					x		x
[13]	x	x	x						x		x
[15]					x						
[17]					x	x				x	
[18]					x					x	
[19]				x						x	x
[20]					x					x	
[21]		x	x	x				x	x		x
[22]	x	x		x		x				x	
[23]					x	x				x	
[25]	x			x		x				x	
[26]	x	x		x		x				x	
[27]					x					x	
[29]								x			x
[30]								x			x
[33]					x					x	
[34]					x						x
[36]								x			x
[41]					x					x	
[45]				x						x	
[47]	x		x	x						x	x

Figure 16: Related Works

cost for initial setup? The example shown is only a concept against maintenance time and does not estimate costs, such as those associated with the labor of updating

documentation, or the cost of implementing a database or some other automated technology. It would be beneficial to know what the long-term cost of setup and upkeep of maintenance through this method has on both small and large systems. Observing the method and its application toward different engineering practices such as agile or open-source development would also be beneficial. Suggested automated application of the method has been provided but these applications require overhead of both database setup and maintenance, or the development of algorithms and storage mechanisms for the documentation. An event-driven architecture designed for maintainability could significantly decrease the effort involved in maintenance, but more research and experimentation is needed to test this theory. Discovering if the cost and additional maintenance of setup for both automated and manual methods will benefit in the long or short term is also desirable. Studies such as these are left for further research.

REFERENCES

[1] Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2013). Software architecture optimization methods: A systematic literature review. *Software Engineering, IEEE Transactions on*, *39*(5), 658-683.

[2] Antonellis, P., Antoniou, D., Kanellopoulos, Y., Makris, C., Theodoridis, E., Tjortjis, C., & Tsirakis, N. (2007). A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering–product quality standard. *Special Session on System Quality and MaintainabilitySQM2007*.

[3] Antoniol, G., Canfora, G., Casazza, G., & De Lucia, A. (2000, February). Identifying the starting impact set of a maintenance request: A case study. In *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European* (pp. 227-230). IEEE.

[4] Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, *36*(11), 81-94.

[5] Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software development practices, software complexity, and software maintenance performance: A field study. *Management Science*, *44*(4), 433-450. 11

[6] Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice*. Addison-Wesley Professional (pp. 117-129). 5

[7] Bell, T. E., & Thayer, T. A. (1976, October). Software requirements: Are they really a problem?. In *Proceedings of the 2nd international conference on Software engineering* (pp. 61-68). IEEE Computer Society Press.

[8] Bengtsson, P., Lassing, N., Bosch, J., & van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, *69*(1), 129-147. 14

[9] Bengtsson, P. (1998, August). Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. In *First Nordic Workshop on Software Architecture (NOSA'98), Ronneby*. 10

[10] Bergey, J., Barbacci, M., & Wood, W. (2000). *Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition: A case study* (No. CMU/SEI-2000-TN-010). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[11] Boehm, B. W., Brown, J. R., & Lipow, M. (1976, October). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering* (pp. 592-605). IEEE Computer Society Press.

[12] Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W. G. (2001). Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, *13*(1), 3-30.

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

[14] Coppit, D., Yang, J., Khurshid, S., Le, W., & Sullivan, K. (2005). Software assurance by bounded exhaustive testing. *Software Engineering, IEEE Transactions on*, *31*(4), 328-339.

[15] Estublier, J. (2000, May). Software configuration management: a roadmap. In*Proceedings of the conference on The future of Software engineering* (pp. 279-289). ACM.

[16] Feiler, P. H., & Gluch, D. P. (2012). *Model-based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley.

[17] Fog Creek Software, Inc. (2000-2014). FogBugz. Retrieved from *www.fogcreek.com/fogbugz*.

[18] Gotel, O. C., & Finkelstein, A. C. (1994, April). An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on* (pp. 94-101). IEEE.

[19] Graaf, B. (2004). Maintainability through architecture development. In *Software Architecture* (pp. 206-211). Springer Berlin Heidelberg. 2

[20] Hayes, J. H., Dekhtyar, A., Sundaram, S. K., Holbrook, E. A., Vadlamudi, S., & April, A. (2007). REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering*, *3*(3), 193-202.

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

[22] Heninger, K. L. (1980). Specifying software requirements for complex systems: New techniques and their application. *Software Engineering, IEEE Transactions on*, (1), 2-13.

[23] Huang, R., Berenbach, B., & Clark, S. (2007). Best practices for automated traceability.

[24] Hunt, J. (2002). Blackboard architectures. *JayDee Technology Ltd*, *27*.

[25] IEEE Computer Society. Software Engineering Standards Committee, & IEEE-SA Standards Board. (1998). IEEE Recommended Practice for Software Design Descriptions. Institute of Electrical and Electronics Engineers.

[26] IEEE Computer Society. Software Engineering Standards Committee, & IEEE-SA Standards Board. (1998). IEEE Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers.

[27] Jarke, M. (1998). Requirements tracing. *Communications of the ACM*, *41*(12), 32-36.

[28] Jorgensen, P. C. (2013). *Software testing: a craftsman's approach*. CRC press. 131-167.

[29] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., & Carriere, J. (1998, August). The architecture tradeoff analysis method. In *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on* (pp. 68-78). IEEE. 12

[30] Kazman, R., Bass, L., Webb, M., & Abowd, G. (1994, May). SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering* (pp. 81-90). IEEE Computer Society Press. 9

[31] Kramer, D. (1999, October). API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation* (pp. 147-153). ACM.

[32] Lethbridge, T. C., Singer, J., & Forward, A. (2003). How software engineers use documentation: The state of the practice. *Software, IEEE*, *20*(6), 35-39.

[33] Mäder, P., Gotel, O., & Philippow, I. (2009, January). Enabling automated traceability maintenance through the upkeep of traceability relations. In *Model Driven Architecture-Foundations and Applications* (pp. 174-189). Springer Berlin Heidelberg.

[34] Marcus, A., & Maletic, J. I. (2003, May). Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 125-135). IEEE.

[35] Michelson, B. M. (2006). Event-driven architecture overview. *Patricia Seybold Group*, *2*.

[36] Matinlassi, M. (2004, June). Evaluating the portability and maintainability of software product family architecture: Terminal software case study. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on* (pp. 295-298). IEEE.

[37] Neill, C. J., & Laplante, P. A. (2003). Requirements engineering: the state of the practice. *IEEE software*, *20*(6), 40-45.

[38] Nii, H. P. (1986). The blackboard model of problem solving and the evolution of blackboard architectures. *AI magazine*, *7*(2), 38.

[39] Nosek, J. T., & Palvia, P. (1990). Software maintenance management: changes in the last decade. *Journal of Software Maintenance: Research and Practice*,*2*(3), 157-174. 7

[40] Oskarsson, Ö. (1982). *Mechanisms of modifiability in large software systems*. VTT Grafiska,.

[41] Ramesh, B. (1998). Factors influencing requirements traceability practice.*Communications of the ACM*, *41*(12), 37-44.

[42] Royce, W. W. (1970, August). Managing the development of large software systems. In *proceedings of IEEE WESCON* (Vol. 26, No. 8).

[43] Shaw, M., & Garlan, D. (1996). Software architecture: perspectives on an emerging discipline. 19-32, 38, 51. 15

[44] Sneed, H. M. (2008, September). Offering software maintenance as an offshore service. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on* (pp. 1-5). IEEE. 6

[45] Sommerville, Ian. *Software Engineering, 5th Edition*. Addison-Wesley Publishing Company, 1996. 8

[46] Sommerville, I. (1996). Software process models. *ACM Computing Surveys (CSUR), 28*(1), 269-271.

[47] Sophatsathit, P. Lessons Learned on Design for Modifiability and Maintainability. 4

[48] Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing. 20