A Proposed Method for Achieving Increased Software Maintainability Through Documentation

Justin Huber
Computer Science
University of North Dakota
Grand Forks, ND 58201
justinhuber86@gmail.com

Abstract

Software maintenance is the final and, potentially, can be the longest and most costly stage in the software engineering process. Software defects; along with changes in business practices, the software operating environment, and customer requirements are all ongoing long after the release of a software system and so it is important for software to evolve alongside them in order to continually meet these needs. 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. Maintainability is discussed and a method for obtaining a higher level of maintainability through documentation is proposed.

1. Introduction

Change is an inevitable and software 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 the software costs time, resources, and funding. Information systems, for example, can use up to 80% of their budget on maintenance costs alone [21]; and over any software's lifetime the cost of [6] change and upkeep can amass to at least 50% of the total system cost [26]. It has been shown [3] that design decisions have an impact on maintenance concerns and can have long lasting and irreversible effects on the software. 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 [25] 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 [25]. Statistics like these show that a balance should be struck between system cost and change, or maintainability. As the difficulty of implementing change raises 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) [16] and the Software Architecture Analysis Method (SAAM) [17], 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 presenting techniques for increasing maintainability in a software system through the documentation from each phase of the development cycle starting with requirements, to code, and finally testing. These three stages continue long after the release of software and are intertwined in a cycle of documentation change alongside system change. The remainder of this paper is organized as follows. Section II and III include a survey of software maintenance, the different variables that affect maintenance, and current methods available for evaluating maintainability in software systems through architecture. Section IV will present a new method for evaluating maintainability, followed by an example in Section V. Finally, future applications, conclusions and current issues or open research questions for the method are provided in Sections VI and VII.

2. Background

2.1 Maintainability

In software engineering two common types of requirements described are functional and nonfunctional [4][26][5][14]. 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 and are also known as quality attributes. Some common quality attributes include performance and availability [4] 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 [8] to maintenance.

Modifiability is about change, the cost of change, and the probability of change [4]. 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. The types of change are explained in a subsequent section. Cost [2] can be split into two types, financial and time. Financial costs are accumulated through labor and hardware or software changes. 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; time and resources spent could have been made on other projects. 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. Finally, modifiability is also concerned with finding the probability of change. Change difficulty and the likelihood change will or will not be required 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 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.

2.2 Types of Change

At its core, maintainability is about change and sustainability. The software development process [26] 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 [17], which were derived [22]. These 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. 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. Feature enhancements are about making additions to, or fixing problems to currently implemented features. This is where most bug fixes would be categorized and where the general term, maintenance, most closely fits.

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.

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.

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 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, and requirements. Software architecture and documentation play an important role in change and are discussed in sections 3 and 4.

3. Maintainability Through Architecture

Software architecture is the "set of principal design decisions made during development and subsequent evolution [28]." 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 [24], as well as quality attributes such as maintainability, performance, availability, security, etc... 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, as certain architectural styles are better suited for different sets of qualities and the four types of change are each affected differently depending on the 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.

3.1. Maintenance Tactics

If nonfunctional requirements, or software quality attributes, are used to measure system behavior set by functional requirements, then tactics [4] 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. Maintainability is concerned with how these components are linked by change. 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 the degree of maintainability.

Cohesion is used to describe how different components 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. Cohesion is important to maintainability in that as the level of cohesion in system components rises, the coupling between them tends to drop since components are have clear, separate goals.

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.

Maintenance tactics are an important tool for measuring maintainability in software systems. The examples above described how tactics affect the software at an architectural and implementation level. Maintainability tactics also play a role at the documentation level and are further discussed in section 5.1.

3.2 Software Architecture 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. There are many methods currently available for evaluating software architectures, two of them are discussed in the proceeding paragraphs.

The Software Architecture Analysis Method (SAAM) [17] 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 has been demonstrated using modifiability as the test, and been shown it to be effective [17].

While SAAM provides a 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) [16] 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; ATAM attempts to accomplish this through a seven step scenario driven process which will not be listed here. The issue with this method as well as SAAM, is that they lacks focus and don't 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.

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) [7], 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) [20]; and many more that are beyond the scope of this paper.

4. Maintainability Through 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 [12] to the longevity and maintainability of software; and lack of proper documentation has been shown [27] 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. Maintainability tactics for architecture should transfer well to the documentation; along with a method for their application in section 5. The following paragraphs detail three areas of documentation: requirements, code, testing, and maintenance requests.

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 [26]. 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 will result in a poorly implemented architecture 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.

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 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, or the original programmer forgets, the way the code is written and what parts it affects. External documentation of code is also desirable. As how the requirements documentation presents the software functionality, external documentation presents an organized and detailed explanation of the source code. External documentation can be created through internal methods, or third-party tools such as JavaDoc [18] 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.

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 [8], so testing documentation is also 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 should to be outlined with a test case that is detailed enough for any tester to perform, and keep track of what changes are being tested, or what additional changes need to be made based on testing. Test documentation, like code commenting, and requirements, 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.

Documentation is an important component of maintainability for 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 section.

5. 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 by reducing the difficulty of making necessary change over time. The previous sections 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. Architecture implements the requirements and code implements the architecture. The requirements document should be recorded first in developing software and the choices made and level of detail presented in them have a lasting effect on the rest of the development process. Poor requirements lead to bad or incorrect design which leads to bad code. Transitively, bad or poorly written code and code documentation is a result of unsatisfactory requirements. Thus, documentation then is an important element to maintainability as the information presented allows for a higher level of understanding of the system's function and structure. The following paragraphs detail a proposed solution to obtaining maintainability through documentation, first by showing how architectural maintenance tactics can be applied to documentation, introducing three new tactics to the current set, and finally detailing how they can be applied to the different areas of documentation.

5.1 Tactics for Documentation

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 pieces or articles 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 articles or pieces grows, so too does the amount of change needed to complete edits to the document. Size for documentation can be a detriment, as the larger the document gets, the harder it may be to understand or find desired pieces. However, this can be offset by a higher level of cohesion and by the amount of detail provided. The original three tactics and their implementation to three levels of documentation, requirements, source-code, and testing, are detailed below along with definitions for understandability, traceability, and networking tactics.

Cohesion, coupling, and size tactics can also be applied to the requirements documentation to promote maintainability, although some in different ways compared to architecture. Much like the architectural components and code, a high level of cohesion is desired for the requirements documentation. Similar functions, or smaller functionalities that are part of a larger function, should be grouped together so they are easy to find and understand when put into context of one another. Cohesion is closely related to how well the documentation is organized, as a well-organized document will more likely exhibit a higher level of cohesion. Requirement coupling can be used as an early detection of maintenance issues that could occur as development of the software progresses. If it is observed that multiple functions are coupled together it could prove worthwhile to rethink the requirements before implementation begins to prevent future coupling problems among components. Highly coupled requirements also create more work once the requirements need to be updated due to change requests, as multiple related requirements may need to be changed as well; which further raises maintenance cost. Finally, size is an important factor for maintainability within requirements documentation. As the size of the documentation increases, so does cost when changes need to be made. However, the level of detail combined with size is what should determine if the documentation is detrimental or helpful to system maintenance. A very large but concise and well detailed requirements document should be more valuable, than a short but poorly written document for example.

Maintenance tactics on code documentation will work slightly different than on the source code itself. Cohesion with comments in code and their subsequent external documentation are tied directly with how cohesive the source code is. Highly cohesive source code will create cohesive external documentation as the generated document will be based on the code presented. Coupling of code comments is undesirable and if the same type of comment needs to be repeated more than once, it should simply be referenced. Finally, when evaluating size, code commenting should be simple but concise as greater detail may make the code harder to read. Extended detail can be woven into the external documentation for more in-depth understanding.

Tactics used on test documentation is similar to the requirements. Test cases should be highly cohesive, in that the case should focus on one function or, more preferably, one piece of that function. As with code commenting, coupling among different test cases creates more work, and if the same test needs to be done in another test it should simply be referenced. Size is dependent on how detailed the testing documentation is. Testing steps and goals should be well defined and laid out in such a way that anyone can run the test, so the greater detail the better. A very small test case with vague documentation will be harder to complete than a large case that is easy to understand and finish.

A fourth maintenance tactic for documentation, understandability [8], 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 such as conciseness, legibility, self-descriptiveness, and consistency need apply [8]. For change to be implemented easily, the documentation involved must be understandable by all those involved, including the stakeholders, developers, testers, and clients. Requirements can split into multiple versions where one is more technical for developers, and another where the information is presented in a less technical manner for stakeholders and other users within the company; a user manual may also be created for customers. Source code documentation should be written so that it is understandable by current and future developers; no short-hand for example. The external code documentation can be written with a mix of technical and non-technical writing so that both developers and QA can understand. Change requests and test cases should be very concise and provide easy understanding as to what needs to be done in order to complete a maintenance request.

The next documentation maintainability tactic 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 [11] [19] [15]. Traceability is used in a variety of engineering concepts and tools such as software configuration management [9], a process for tracking changes in code versions; and third party change request, bug, and test case repositories such as FogBugz [10] 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 [13]. A documented history of change within each level of documentation should provide both higher understandability as well as future ease of maintenance.

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 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 history tracking. What this provides is a quick reference lookup between all documents for any individual requirement, test case, or code implementation. In this way, there should be less time spend going from requirements checking, to code implementation, to test case generation or editing; as well as less time spent attempting to understand all assets of a particular requirement, piece of code, or test case. Internal networking requires that each item of documentation, 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 two documentation levels. External networking creates a link between the requirements, code, and testing documents.

5.2 Applying Documentation Tactics

This section will describe the proposed steps for achieving a higher level of maintainability. There are six steps in total, each applying the different maintainability tactics detailed in the section 5.1: cohesion, coupling, size, understandability, traceability, and networking. The following paragraphs will detail these six tactics and describe how they can be applied to the requirements, source-code, and testing levels of documentation either during or after development.

First is the application of cohesion. Cohesion in the requirements document is about organization and keeping to the adage "a place for everything and everything in its place". The requirements document as a whole should fit within designated sections each with different goals, such as functions, third party tools, operating environments, business practices, glossary, etc.; with each individual listing in the document organized in the same manner. Individual entries should also adhere to the same rules as architectural level cohesion, in that they only entail information that is relevant to them. 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 be split into appropriate sections such as by requirement category, or design component.

Second is coupling and is concerned with direct ties between requirements, testing, and code that will require parallel change with another entry in that document category. The more coupling there is between different parts of each section or different sections, 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.

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 shorter 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 in making required changes. In general, more comments can only help in understanding the code, unless written in poorly documented short-hand technical terms (see step 4). Shorter sections of code with no comments could potentially be harder to maintain than larger sections of code with highly detailed comments. Small test cases can become harder to perform if the test description and steps are vague.

Step four is ensuring understandability. Documentation is only as good as the way it is presented for whom it was created. Code commenting and external documentation 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 [14]. 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.

Step 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 [15] [23]. The addition of a history log within requirements, 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. Individual parts of documentation could also include their own history. Source-code commenting, for example, could comments where code was updated, or at the top of the class with a list of changes as they have occurred. A 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 goes through 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 step is the networking among documents. Requirements that are realized in code should be linked, and test cases created for those requirements should be linked as well. The goal is to ensure that each document type has a way to easily find a corresponding entry in another document. A typical change request starts with the request for change, followed by a validation of the requirements 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 [26]; this is presented in figure 1a. Document networking requires that 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 is made there; this is shown in figure 1b.

The simple way to realize networking between documents is to establish a library type system where each item (line, function, or class code, entry in requirements, 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 two 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 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 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 analyzing 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 could provide the same results. Standard architecture maintenance tactics 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.

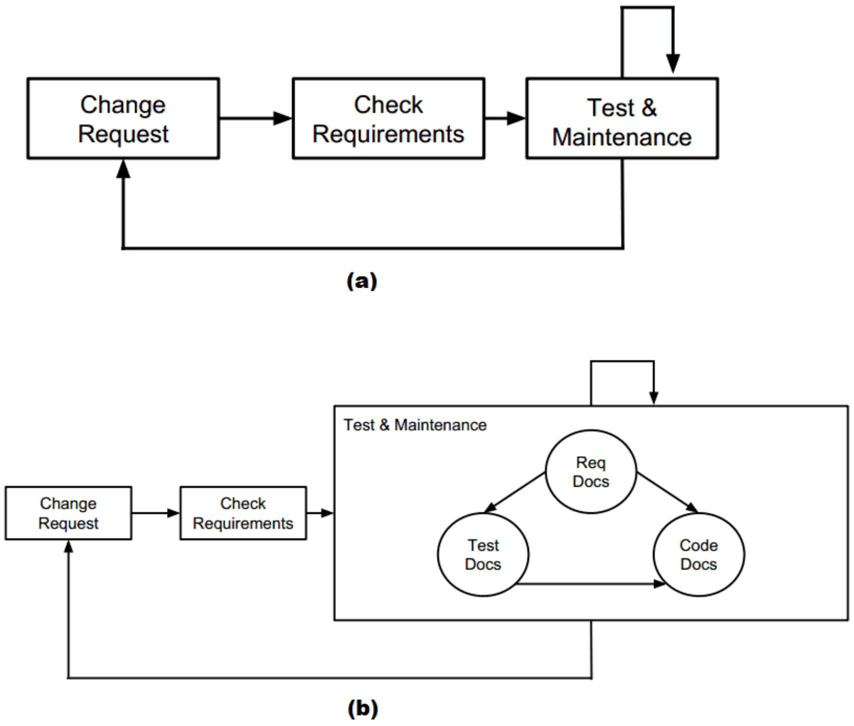

Figure 1: Maintenance Cycle

6. Conclusion

Maintainability evaluation and tactics 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, but with the definition of system requirements within the Requirements document. 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, ultimately ending in low maintainability causing higher costs on an already costly area of software development.

A proposal for increasing maintainability through the documentation 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. Tactics offer a direct correlation with change and the ease in which change can be implemented, while networking and traceability adds yet another layer of ease when trying to not only update change throughout all documents, but understand current and past versions of software. An example has not been provided as development of this proposal is still ongoing and part of continuing research by the author.

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] 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.

[3] 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

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

[5] 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.

[6] 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

[7] 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.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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

[16] 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

[17] 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

[18] 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.

[19] 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.

[20] 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.

[21] 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

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

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

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

[25] 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

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

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

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