

Reconstructing Software Architecture Documentation for Maintainability

by

Jonathan Witkamp

Reconstructing Software Architecture Documentation for Maintainability

by

Jonathan Witkamp

A thesis presented to the University of Amsterdam
in fulfillment of the thesis requirement for the degree of
Master of Science
in
Software Engineering

Thesis Supervisor: Drs. H. Dekkers
Department Supervisor: Prof. Dr. P. Klint
Internship Supervisor: Drs. J. Meijerink (Lost Boys)

Amsterdam, The Netherlands, 2006

© 2006 Jonathan Witkamp

Abstract

Software development is a complex and expensive undertaking. After the development of the software, the software enters the maintenance phase. Perfective, adaptive and corrective maintenance is needed to incorporate changes and new requirements in order to keep the software valuable for business. Unfortunately, the maintainability of software is frequently overlooked during the development of new systems in the software industry. The Content Management System (LBCMS) of Lost Boys is a typical example of this disposition. Enhancing the maintainability of a software system such as the LBCMS offers a significant payback opportunity, because maintenance costs can utilize up to 85% of the total cost of ownership of a software system. In addition, the correctness of changes and the quality of the design affect the quality of the software. If the maintainability of a system is poor, deterioration of the software will increase and the system rapidly loses its value for stakeholders.

There are no straightforward solutions for the maintenance problems at Lost Boys. This research proposes a solution for enhancing the maintainability by retrieving and documenting architectural knowledge of the LBCMS. The solution is based on the philosophy of software architecture and requirements engineering. However, most approaches for designing software architecture or capturing requirements assume that the system is yet to be build. Approaches for retrieving architectural information from established systems tend to focus on source code. Although source code contains architectural evidence, it is the result of certain design decisions and constraints. In this research, the approach for retrieving architectural knowledge for maintainability combines several approaches in order to understand the maintenance problems of Lost Boys from the perspective of stakeholders involved, to understand the architecture and capabilities of the system, and to effectively present the architectural knowledge in documentation in order to enhance the maintainability of the LBCMS.

The developed method to obtain the documentation consists of four phases; problem analysis, understanding requirements, understanding the architecture, and documenting the architecture. The techniques used in the approach are; interviews with stakeholders, stakeholder analysis, problem analysis, review workshops, source code browsing, and computer aided metrics.

The results of this research reside in the software architecture documentation of the LBCMS, in which the system is described from the perspective of business goals, stakeholder concerns, features, quality requirements, architectural views, and design decisions. A great effort is made to develop documentation in compliance with the information needs of stakeholders. In addition, the documentation has attributes which make the documentation itself maintainable, such as versioning, tracing, and verification. The correctness and completeness of the documentation is validated by means of interviews, system testing and a review workshop. The applicability of the documentation is validated by means of two personas, which represent two stakeholders at Lost Boys. In addition, an architectural strategy is developed as a guideline for Lost Boys for future improvement of the maintainability of the LBCMS. The approach and results are successful with respect to the goals of this research, and can be used in other projects to enhance a systems support for maintainability.

Table of Contents

Abstract	v
Chapter 1 Introduction.....	1
1.1 Background & Context.....	1
1.2 Motivation for this Research	4
1.3 Research Goal.....	4
1.4 Research Questions	5
1.5 Structure of this Thesis.....	5
Chapter 2 Software Maintainability and the Role of Documentation	6
2.1 Introduction	6
2.2 The Evolution of Software	6
2.3 Software Quality.....	7
2.4 Software Maintenance	8
2.5 Software Documentation	11
2.6 Conclusion.....	14
Chapter 3 Problem Analysis.....	15
3.1 Introduction	15
3.2 The problem description according to Lost Boys.....	16
3.3 The causes & effects.....	16
3.4 Why is this a problem.....	17
3.5 The problem from the perspective of John Foo; a Software Engineer	18
3.6 The problem from the perspective of Miranda Bar; a Technical Consultant	20
3.7 The impact of the problem	20
3.8 Possible solutions	21
3.9 Hypothesis	23
Chapter 4 Approach for retrieving architectural knowledge	24
4.1 Existing methods for retrieving architectural knowledge.....	24
4.2 Approach.....	25
4.3 Understanding the Requirements of the LBCMS.....	25
4.4 Understanding the Software Architecture of the LBCMS.....	26

4.5 Validation of the results	27
Chapter 5 Understanding Requirements: Software Requirements Specification.....	29
5.1 Approach to Requirements Engineering	29
5.2 Stakeholders and their Concerns.....	30
5.3 Business Goals	31
5.4 Feature Requirements	33
5.5 Quality Requirements	34
5.6 Conclusion	36
Chapter 6 Understanding Architecture: Software Architecture Documentation	38
6.1 Approach.....	38
6.2 Understanding the internal structure	39
6.3 Design Decisions	41
6.4 Architectural Views	42
6.5 Maintainability Tactics	46
6.6 Conclusion	47
Chapter 7 Results and Conclusion	48
7.1 Validation of the documentation.....	48
7.2 Assessment of the Applicability of the Documentation.....	50
7.3 How the documentation supports the tasks of John Foo and Miranda Bar.....	52
7.4 Discussion.....	53
7.5 Conclusion	55
References.....	57
Appendix A Software Requirements and Architecture Documentation	60
Appendix B Workshop presentation.....	61

Table of Figures

<i>Figure 1 the structure of the LBCMS websites.....</i>	<i>2</i>
<i>Figure 2 Context diagram of the CMS and its surroundings</i>	<i>3</i>
<i>Figure 3 A Staged Model for the Software Life Cycle (adapted from [Rajlich 00]).....</i>	<i>7</i>
<i>Figure 4 the focus of information needs during development and maintenance.....</i>	<i>11</i>
<i>Figure 5 software development and documentation.....</i>	<i>12</i>
<i>Figure 6 Cause and effect analysis</i>	<i>17</i>
<i>Figure 7 a computer generated module view of the LBCMS (Eclipse Inspector Gadgets).....</i>	<i>18</i>
<i>Figure 8 approach for retrieving relevant knowledge</i>	<i>25</i>
<i>Figure 9 approach to requirements engineering.....</i>	<i>29</i>
<i>Figure 10 Feature description.....</i>	<i>34</i>
<i>Figure 11 Quality grid</i>	<i>36</i>
<i>Figure 12 Quality Requirement description.....</i>	<i>36</i>
<i>Figure 13 Software Architecture Development phase.....</i>	<i>39</i>
<i>Figure 14 primary view of the metrics view</i>	<i>41</i>
<i>Figure 15 Component and Connector view.....</i>	<i>44</i>
<i>Figure 16 Allocation view</i>	<i>45</i>
<i>Figure 18 a Maintainability strategy with tactics</i>	<i>46</i>
<i>Figure 19 Reasoning about maintainability tactics</i>	<i>51</i>
<i>Table 1 Captured stakeholders.....</i>	<i>30</i>
<i>Table 2 Captured business goals Lost Boys</i>	<i>32</i>
<i>Table 3 List of LBCMS features</i>	<i>33</i>
<i>Table 4 Major design decisions.....</i>	<i>41</i>
<i>Table 5 Communication needs served by architecture.....</i>	<i>42</i>
<i>Table 6 rationale for chosen views.....</i>	<i>43</i>

Chapter 1

Introduction

Software requirements and software architecture are well recognized fundamentals of modern software development. Almost all software systems are developed with certain requirements in mind, and every software system has a software architecture. The iterative and collaborative nature of software development and maintenance activities results in an increasingly need for explicit knowledge of the requirements and the architecture of the system. This knowledge can be captured in documentation and serve as a communication vehicle among stakeholders to manage complex dependencies and interactions [Bass 03, Kruchten 95]. The capturing of the architectural knowledge and the software requirements to support the maintainability of a three year old software system is the topic of this research.

This chapter explains the motivation for this research and presents the research goal and questions. This chapter is concluded with a brief overview of the structure of this thesis.

1.1 Background & Context

Lost Boys

Lost Boys Amsterdam is a *full-service internet company*; Lost Boys delivers innovative marketing solutions and offers an integrated approach of business and branding strategy consultancy, combined with marketing, communications and information technology. Traditionally, Lost Boys develops marketing oriented communication strategies for customers, using the internet as primary medium. The software engineers select and implement a commercial of the shelf (COTS) product that fits the needs of the customer. The visual and interaction designers create the interface of the system. These systems are meant for facilitating the delivery of content using internet and are widely known as Content Management System, or CMS for short. The next section will explain a CMS in more detail.

Recently, Lost Boys acquired a company named Impact in order to expand their services with, among other things, a Content Management System (CMS). Impact initially developed the CMS for Mitsubishi Motors Europe (MME), but the aim was to make the system flexible in order to implement the system for other customers with different requirements. With flexibility as a major concern, the design of the system comprises several frameworks which are believed to support future changes and scalability to enterprise usage. At this moment, Lost Boys implemented the CMS for three customers.

The Content Management System of Lost Boys provides the desired features for customers to manage their content. However, it does not meet all quality requirements, such as modifiability, portability, testability and maintainability. These quality requirements are important for Lost Boys and its customers since maintenance is an increasingly valuable undertaking.

Lost Boys employees often express their need for technical documentation of the LBCMS in order to keep the LBCMS viable for business. The goal of the documentation is to make knowledge transferable, accurate, and manageable. How this can be achieved is discussed in this thesis.

LBCMS case: Content Management Systems

A Content Management System such as the LBCMS is a software system for facilitating and organizing collaborative management of content. But what does this mean exactly? A working definition of content is:

On internet (a webpage), content is material of interest, put there and changed frequently in order to encourage visits to that page, possibly producing revenues.

Content can be digital text, images, sound, animations and the like. A working definition of content management is:

A set of processes and technologies that support the evolutionary life cycle of digital information (content).

In the context of this research, a Content Management System (CMS) is defined as:

A computer software system for facilitating and organizing collaborative management of content.

The LBCMS was initially built for Mitsubishi Motors Europe (MME). The LBCMS enables users in different roles to publish and manage content on an international, national or local website. The users have different roles concerning content management; some users are responsible for publishing corporate content, such as annual revenues, images of new products and prices. Other users are responsible for publishing content in a specific language for a specific market, while using corporate content.

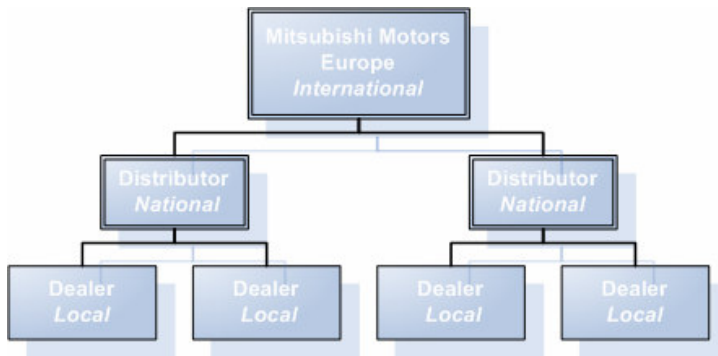


Figure 1 the structure of the LBCMS websites

The system encompasses multiple websites for different audiences. There is a corporate website which provides information about the company for guests and distributors. Every distributor in Europe has a localized website, providing information about the company, products and the dealers. A dealer sells products and has its own website, which contains information about products, prices and showcases. Figure 1 illustrates the structure of the website in the LBCMS. All websites are based on templates which are developed by Lost Boys, so all websites have the same look & feel and represent the corporate identity of the customer. The websites should be available in all countries in Europe; for clients of MME as well as for the content managers, 24 hours a day.

With “high” performance and “high” availability as important requirements, several technical measures have been taken to satisfy these requirements, such as hardware redundancy, load balancing, and monitoring. In order to make the system flexible for future changes, the decision was made to implement the system using Java and Enterprise Java Beans (J2EE). Also, popular techniques such as Struts, Maven, and XSLT are used within the software in order to separate concerns with respect to data persistency, business logic, user interface, and the deployment process.

The Scope of the LBCMS

Figure 1 illustrates the context of the LBCMS against its surroundings. The LBCMS is the big box in the middle and the arrows pointing to users or other systems represent the interfaces of the LBCMS.

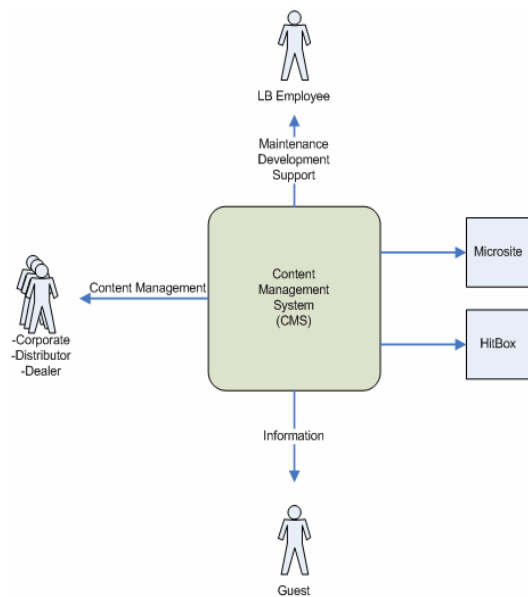


Figure 2 Context diagram of the CMS and its surroundings

- *Guest*: The product is used by guests, who seek information about products, events, contact information, and other content which is published by the customer (MME) of the LBCMS.
- *Customer*: The product is also used by the customer (MME in this case) to publish content on the websites (corporate, national and dealer).
- *Lost Boys employee*: LB employees have certain tasks to perform with the LBCMS and therefore have different interfaces, such as Eclipse for Java programming and Enterprise Manager for database management.
- *Microsites*: Microsites are small promotional websites which are developed and deployed separately.
- *HitBox*: statistics about the use of the system are collected using an external tool, HitBox.

1.2 Motivation for this Research

During the development of a software system, design decisions, trade-offs and dependencies between requirements, and organizational constraints shape the architecture of the system as it converges to its operational state. This knowledge is important when changes need to be made in the system during the maintenance phase of the software life-cycle¹. Knowledge can be captured in documentation, but documentation is often incomplete, neglected or poorly maintained during the development and maintenance phase [Bennett 00, Parnas 01]. Important architectural drivers remain implicit and are easily misunderstood by maintenance engineers. Consequently, the absence of relevant architectural knowledge makes maintenance an obscure and unpleasant activity, the cost of maintenance increases, and errors are more likely to occur when changes are made.

This is also true for the Content Management System (LBCMS) of Lost Boys Corporation, which was the subject of this research. Developers at Lost Boys need appropriate documentation to assess the feasibility, impact and viability of changing the LBCMS during maintenance. The motivation for this research springs of this need of Lost Boys to improve the documentation of the system. The improved documentation should provide knowledge which enables stakeholders to reason about the system in order to enhance the systems maintainability. But what should be documented and how? This question is answered by combining several techniques proposed by literature and observations made by the author. The results are assessed in interviews and a workshop, in which stakeholders reasoned about the system based on the documentation.

1.3 Research Goal

The goal of this research is to develop documentation of the LBCMS to obtain explicit and transferable architectural knowledge of the system. The stakeholders of the LBCMS have different stakes in the LBCMS, which result in differing communication needs [Bass 03, Kruchten 95, Lauesen 02]. The documentation should take these stakes and communication needs into account. Research is done to clarify what knowledge should be captured, and how this knowledge should be presented in documentation. Ultimately, the documentation can be used by stakeholders to assess the impact and feasibility of changes to the LBCMS software, before the actual changes are made. This is necessary in order to lower the costs of maintenance and make the maintenance activities easier. However, it is not feasible to assess whether this ultimate goal is achieved within the timeframe of three months, so the focus of this research is on communication needs and accurate architecture documentation.

¹ See chapter 2 and 4 for more information about the maintenance phase and the information needs.

1.4 Research Questions

The following research questions are formulated based on initial discussions about the project, the problems which initiated the need for this research are analyzed in chapter 3:

1. *What maintenance problems need to be solved and how can software architecture documentation solve it?*

The starting point of this research is to explore the problem space and define which and how the problems can be solved by capturing knowledge in documentation. So to answer this research question, it is imperative to understand the characteristics of maintainability and its difficulties. Architectural knowledge can be captured in documentation, but what documentation is relevant for maintenance? In addition, it is necessary to determine the information needs of the stakeholders involved.

2. *When is the architecture documentation of the system satisfactory?*

It is important to determine if the captured knowledge is correct and complete enough to reduce the impact of the maintenance problems. How big are these problems? What is the impact of the problems? What are the information needs of the stakeholders? What is missing?

3. *How is knowledge of the system retrieved in order to capture it in documentation?*

Several approaches are available to retrieve knowledge, but what approach is applicable given the LBCMS case?

1.5 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 discusses the characteristics of a software system subject to maintenance, and what knowledge is worth capturing to support the maintenance phase (“*what*”). The information presented so far is used in Chapter 3 to analyze the maintenance problems concerning the software of Lost Boys. The analysis resulted in a hypothesis to answer the second research question (“*when*”). An approach is presented in Chapter 4, which answers the third question (“*how*”). Chapter 5 describes the retrieval and capturing of the requirements specification and Chapter 6 describes the retrieval and development of the architecture documentation. In Chapter 7 the results are validated and the conclusions of this research are presented.

Chapter 2

Software Maintainability and the Role of Documentation

2.1 Introduction

In this chapter, Software Maintainability and the documentation which supports the maintainability of a software system are discussed. First, the characteristics of the life-cycle of a software system are described. It is important to understand that software evolves over time. Second, the consequence of software evolution on the quality of software is discussed. In addition, maintainability is defined as a quality of software and the consequences of the evolution and maintenance are discussed. Finally, the role of documentation is described; what knowledge is necessary to support the maintainability of a software system and how can this be documented?

2.2 The Evolution of Software

Parnas (2001) pointed out that deterioration of evolving software is a growing concern over time. Two different factors contribute to the corrosion of software; changes made in the system, and needed modifications which are never made. If nothing is done to deal with this phenomenon Parnas called *software aging*, the internal structure of a system will decline, more bugs and failures will occur, and the system will be harder to maintain. Fortunately, there are possibilities to reduce the impact of aging; one important step to accomplish this is enhancing the quality of the documentation [Fowler 99, Parnas 01, Bennett 00].

Software Aging

“Programs, like people, get old. We can ‘t prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable”
David L. Parnas

A successful software system inevitably evolves over time. The evolution of software systems during the software lifecycle originates from user-generated requests for change, and improvements and changes made during maintenance activities [Bennett 00]. The evolution of software is accompanied by an increasing size and complexity of the software, because of changes made and the incorporation of new features. This makes the software harder to comprehend and maintain.

[Rajlich 00] argues that the post delivery phase of the software lifecycle consists of several stages, which are illustrated in Figure 3. After the *initial development stage*, the software enters the *evolution stage*. During the evolution stage, the software will generate its largest reimbursement for the sponsor of the system. Also, a lot of efforts are made to maintain the system in order to satisfy new requirements and to resolve issues. It is estimated that the cost of the evolution stage can utilize up to 60% of the total cost of ownership (TCO) of the software system. When the software becomes too large, complex, or when knowledge about the system is lost, the software enters the *servicing stage*. Only critical patches are implemented in order to keep the software running. The eventual *phase-out stage* of the software life-cycle becomes inevitable.

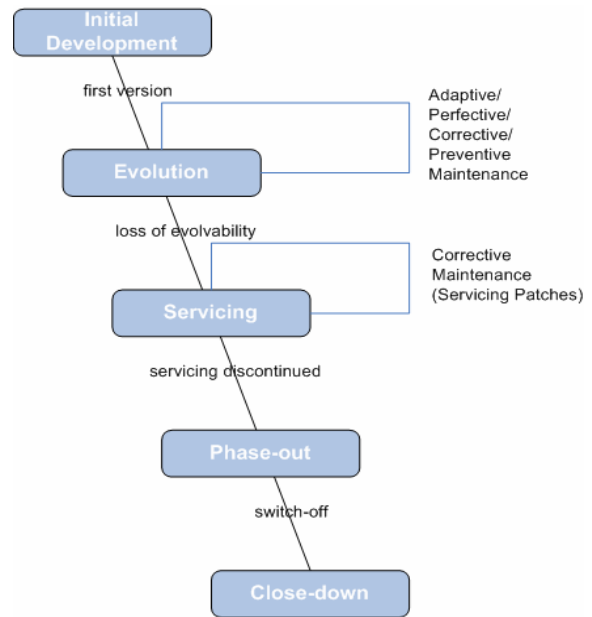


Figure 3 A Staged Model for the Software Life Cycle (adapted from [Rajlich 00])

Consequences of Software Evolution

[McConnell 04] argues that a distinction can be made between two kinds of consequences of software evolution:

- Improvement of software quality; modifications are treated as opportunities to enhance the software design and the support for quality.
- Degrading of software quality; modifications are treated with superstition and as bandage to cover up errors. And, as [Parnas 01] pointed out, deterioration of software occurs when modifications are needed, but never made.

This implies that the consequences of software evolution can be manipulated, towards either improvement or degradation. A key to improvement instead of deterioration is the ease with which the system can be maintained (*maintainability*). However, there are consequences of evolution which are inevitable, such as the growth of the amount of code, increasing complexity, and the increasing probability of failures and errors [Bennet 00, McConnell 04, Parnas 01]. In order to improve the software quality and manage deterioration during the evolution stage, it is imperative to support the maintainability of the software system.

2.3 Software Quality

Software evolution has consequences on the quality of software. The software quality of a system is defined by its support for certain quality attributes, such as maintainability, modifiability, security

[Bass 03]. Software quality is also known as the nonfunctional part of a system, because it defines *how well* a system must perform its tasks, not *how* [Lauesen 02]. For example, a document must be sent to a predefined printer within ten seconds. This *quality requirement* concerns the performance of a system and says nothing about how printing is carried out by the user.

The quality attributes have a strong cohesion. Some attributes support each other, while others are contradicting. For example, the ease of changing the system (modifiability) and the testability of a system both support each other and the maintainability of the system. Security measures have negative influences on performance, because techniques like encryption and authentication need to be computed by the system. It is widely accepted that it is impossible to achieve all quality goals simultaneously [Bass 03, Lauesen 02, McConnell 04]. This means that trade-offs need to be made among the quality attributes. In order to make the right decisions, it is necessary to analyze quality requirements of the system. We ask questions like: *is a more secure system more important than the performance? What does secure mean in the context of the system, when are our security goals achieved?* The answers to these questions result in quality requirements, which need to be precise and measurable [Bass 03].

Characteristics of Quality Attributes

David McConnell (2004) makes a distinction between *external* and *internal* quality attributes. He defines external quality attributes as characteristics of the software that a user is aware of. The usability (ease of learning the system) is an example of an external quality attribute. As is reliability: the ability of a system to execute functions according to the requirements and without failures. This is observable behaviour which users care about. Internal quality attributes are defined as characteristics which users do not observe directly, but are of equal importance because they also have an impact on the software quality. For example, maintainability depends on the understandability of the system; is the code well structured and comprehensible? Examples of internal quality attributes are: maintainability, modifiability, testability, flexibility. During this research, it became clear that the focus of this project should be on the internal quality attribute *maintainability* of the LBCMS.

2.4 Software Maintenance

Many studies emphasize the need for careful considerations concerning the maintainability of a system. Yet, maintainability is still a poorly defined quality attribute and not widely accepted in the software industry [Bennett 00]. Software maintenance is defined in [ISO 95] as:

The software product undergoes modifications to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity.

An earlier, less comprehensive definition was given in [IEEE 93]:

The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

The post-delivery nature of maintenance becomes clear in both definitions. Both definitions relate maintainability to modifications in the software in order to improve, change, or correct certain attributes or artifacts. This *process* results in a modified environment, while the conceptual integrity of the system should be preserved. The quality attribute *modifiability* (ease of change) of the software is mentioned in both definitions. Modifiability affects maintainability, which on its turn affects the *maintenance process*. An interesting point is mentioned in the first definition about documentation as an important artifact when it comes to maintenance. What is missing in both definitions is the ability to verify modifications (testing for correctness) and the ease of understanding the system.

Software maintenance can be described as a service, whereas software development can be described as product development [Niessink 00]. The difference between the service perspective versus the development perspective is the perception of the customer. The customer cares about the results of the development efforts and not about the development itself. The customer does care about how an issue is received, processed, and implemented during maintenance. The customer perceives maintenance as a service to improve or modify a system.

If maintenance is the longest phase of the software life-cycle, why are the qualities that support maintenance often overlooked in software development organizations? Software development is an expensive undertaking and essential in order to produce an income. Therefore, most software development organizations focus on the initial development of the software. However, the costs of a system in the evolution stage of its life-cycle (in which most maintenance expenses are made [Rajlich 00]) will increase over time, and eventually exceed the cost of development.

Categories of Software Maintenance

In [Lientz 80], maintenance is divided in four categories:

- Adaptive maintenance; changes in the software environment
- Perfective maintenance; new user requirements
- Corrective maintenance; fixing errors
- Preventive maintenance; prevent problems in the future.

In [IEEE 90], maintenance activities were defined in accordance of the definition given by [Lientz 80], except for the fact that preventive and perfective maintenance categories are merged. So the maintainability of a system consists of three characteristics; *adaptive*, *perfective* and *corrective* maintenance. Adaptive maintenance has a strong relationship with the quality attribute *portability* of software. Perfective maintenance relates to the *modifiability* of the system [Bennett 00]. Requirements should be managed in order to modify the system with more confidence [Wiegiers 04]. Corrective maintenance depends on the *testability* of the system [McConnell 04]. Most problems with

maintenance are most likely the result of perfective maintenance issues; changing user and quality requirements [Lientz 80]. The *understandability* of a system supports all maintenance categories.

The Cost of Maintenance

Lientz and Swanson (1980) pointed out that approximately 85% of all maintenance costs come from adaptive and perfective maintenance efforts. The costs of maintenance are the sum of the resources needed to maintain a system. This includes software, hardware, personnel, and time. The impact of a change in the system can also increase costs; when changes are treated as hacks, deterioration will occur and the costs will increase over time. The costs can increase even more if a system is not comprehended by maintenance personnel, due to the complexity of the software or the absence of documentation [Parnas 01]. In addition, the correctness of changes and the impact of changes on the design contribute to the costs; if changes can not be verified and the change has a big impact on the system, costs will increase. Especially when the impact of the change is not assessed before the change is made.

Software architecture documentation is accepted as an effective tool to reduce maintenance costs, by providing transferable knowledge about design decisions, qualities, features and the internal structure of the system [Bass 03, Bennet 00, Fowler 99].

Maintainability Decomposed

In [IEEE 90], the maintainability of a software system is defined as: *the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment*. Based on this definition, it is reasonable to state that maintainability of software has a strong tie with the software quality attributes modifiability, testability, portability and understandability. In this section, these quality attributes are discussed in more detail.

Modifiability concerns the ease by which changes can be made and the costs of these changes. According to [Bass 03], the modifiability of a system brings up two concerns:

- What is subject to change? This affects the internal structure of the system.
- When is the change made and by whom? This affects the organization and processes responsible for maintaining the system.

Testability also contributes to the maintainability of a system. If solid test procedures are in place, such as integration testing and acceptance testing, a modification can be made with more confidence and is error prone. In [McConnell 04], testing is defined as an instrument to verify if requirements are met.

Portability is the ease of modifying a system to an environment which differs from the environment for which the system was initially designed [McConnell 04]. At Lost Boys, each developer needs to obtain an instance of the code of the system from the code repository. To verify if modifications are correctly made, a developer needs to compile the system into an executable version and run the system on his or hers developer pc.

Understandability of a system is the ease of which a developer can comprehend the structure of the system and code. If the structure of the system or its code is not understood by developers, maintenance will be difficult if not impossible to execute. In literature, documentation is thought of as an essential artifact to support understandability [Bass 03].

Information needed to perform maintenance

When a maintenance engineer needs to perform his tasks, a significant amount of time is spent on understanding the system [Fjelsted 79]. An interesting observation is made in [Walz 93]; during the initial development of a software system, the focus of the development team is on relevant domain and technical knowledge of the team members. Requirements become increasingly important throughout the project and when the system converges towards implementation, the design (architecture) becomes the focus of attention. During the maintenance phase, the information needed to perform maintenance tasks is on design, requirements, and domain and technical knowledge respectively. So the focus of needed information during maintenance tasks is reversed compared to development (see figure 4).

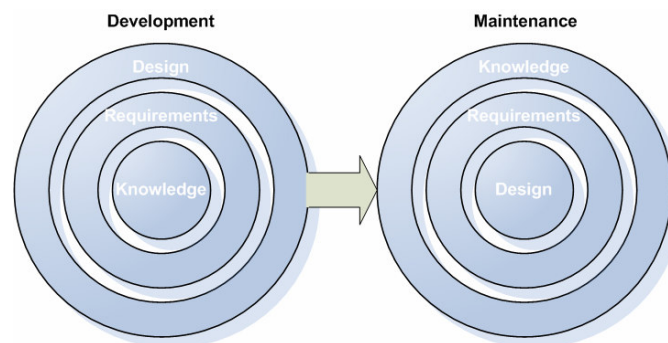


Figure 4 the focus of information needs during development and maintenance

Design information is needed to understand the internal structure of the system and determine where changes should occur [Bass 03]. Bosch (2004) pointed out that architectural design knowledge is lost as a system evolves and that the need for this knowledge is underestimated. The need for design and requirements information is a rationale for capturing design and requirements knowledge in documentation. The information needs of important stakeholders at Lost Boys are discussed in more detail in Chapter 3.

2.5 Software Documentation

The development of software systems comprises several phases. These phases include Requirements Engineering, Architecture Design, Construction, Testing, Deployment and Maintenance. As described earlier, maintenance is the last phase before a system is phased out and closed down. It is widely recognized that the phases before maintenance need to be done in order to make maintenance possible. The products produced during these phases, such as a Software Requirements Specification (SRS) and a Software Architecture design, are valuable artifacts for

maintainers. These artifacts help to comprehend the system by providing information about requirements, design decisions, and the internal design of the system. Research pointed out that of all maintenance tasks, most time is spend on understanding the software [Fjelsted 79]. The purpose of software documentation such as software architecture documentation is to support software maintainers with that expensive understanding [Glass 89].

Types of Documentation

It is important to determine what documentation is useful for maintainers of a software system. Glass (1989) warned the software industry by saying that too much documentation is just as bad as no documentation at all. So it is important to survey what types of documentation are useful for maintenance activities. The next chapter explains why software architecture documentation is useful for Lost Boys, based on the problem analysis. First, types of documentation which ideally are produced during the development of a system are discussed. Figure 5 illustrates the phases of a typical development process according to [McConnell 04], with corresponding documentation.

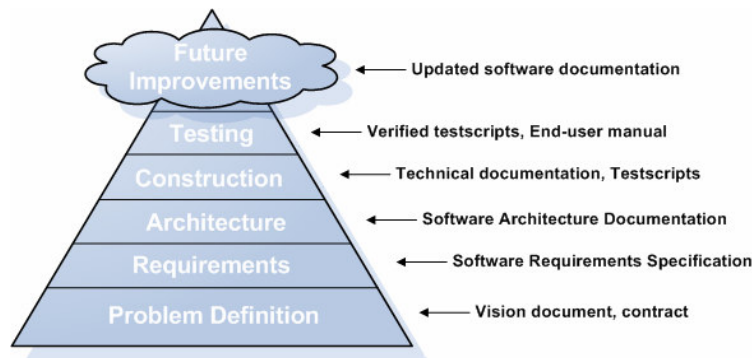


Figure 5 software development and documentation

Types of documentation include:

- Software Requirements specification: documented business goals, project drivers, functional- and quality requirements, issues and risks [Lauesen 02].
- Software Architecture documentation: provides an overview of the internal structure of software. It includes relationships with the environment (context) of the system and construction principles to be used in design of software components [Bass 03].
- Technical documentation: documentation of code, algorithms, interfaces, and APIs.
- End User documentation: manuals for the end-user, system administrators and support staff.
- Testscripts: documentation to verify if the requirements are implemented acceptably.

The applicability of documentation

An empirical study discussed in [Lethbridge 03] illustrates how software engineers use software documentation. A survey was conducted in which the respondents classified the effectiveness of the

available software documentation for certain tasks. More than fifty percent of the respondents found the available software documentation effective for tasks which involve learning, testing, or working with a system. Half of the respondents found documentation effective when knowledgeable developers are unavailable or when “big-picture” information is required. Only one-third of the respondents found the documentation effective for maintaining a system. Why? [Bennett 00] points out that updating documentation is often neglected during development and maintenance. As a result, the available documentation becomes less valuable as a reference for future maintenance tasks. This problem can be reduced by emphasizing the need to maintain the documentation, but also by writing documentation which provides an abstraction of the system in which “low-level”¹ changes are less relevant. A system can support future changes, without affecting the architecture of the system [Bass 03]. Hence, the documented architecture provides a “big-picture” overview of the system, without the need to change the documentation when a “low-level” change is made. The applicability of documentation depends heavily on which information is captured and how this information is presented. In addition, the documentation should be maintained, and therefore maintainable.

The role of documentation for maintenance

There are several forms of documentation which support maintenance activities; a requirements specification is useful for assessing requirements and to make trade-offs. An architecture specification is a representation of an engineered system. It provides a “big-picture” overview, because it represents information about system elements, the relationships among those elements, and the rules governing those relationships [Bass 04]. Comments in code are considered good documentation as well, as long as certain standards are applied [McConnell 04]. Documentation has several important roles in maintenance:

- To offer understanding of a system, and how the system can be exploited [Lauesen 02]. This is achieved by providing information about the goals and requirements of a system, how they relate to one another and what trade-offs are made during development and maintenance.
- To serve as a communication vehicle among different stakeholders and expresses their concerns [Bass 03, McConnell 04]. Documentation is ideally developed with incorporation of the concerns and information needs of stakeholders.
- It serves a transferable abstraction of a system [Bass 03], because a system is described from a certain high-level perspective without revealing too much obscure details. It is transferable because documents can be easily transferred through, for example, a network, e-mail, or hardcopy.
- It encompasses valuable design knowledge of a system [Kruchten 95]. Design decisions accompanied with their rationale provide information for maintainers in order to assess impact of changes and understand why the architecture is the way it is [Bosch 04].

¹ “Low-level” changes include modifications to algorithms, calculations, interface assets, database records

- To support evaluation in order to assess implications of changed requirements [IEEE 93].

Although a lot of knowledge can be captured in documentation, it is imperative to assess which types of documentation is desired. Too much documentation can obfuscate the knowledge it encompasses. In addition, one should accept that not all relevant knowledge can be captured. Some knowledge is already lost during the evolution of the system; deterioration has already occurred, key-personnel left the project, and information remains tacit despite all the available techniques to elicit it.

The Role of Documentation for Requirements Management

Perfective maintenance is about changing requirements [Lientz 80]. An anonymous writer once wrote “Software requirements are like water; it’s easier to build on when frozen”. Changing requirements are a concern for software developers, because the impact of changes can utilize up to 70% of rework [Wiegiers 04, McConnell 04]. In [Wiegiers 04], requirements management is defined as an activity consisting of four tasks: change control, version control, assessing requirements for feasibility and impact (time, resources, correctness, quality) and status tracking. These tasks can be supported by documentation; the requirements need to be specified, measurable, and manageable. The relevance of the documentation for requirements management is discussed in chapter 6.

2.6 Conclusion

In this section, the findings so far are briefly summarized. The problems at Lost Boys with respects to the LBCMS are investigated in depth in the following chapter.

- Software evolution has a consequence on software quality. Managing the evolution of the system slows the aging process and extends the time in which the system is viable and profitable.
- The maintainability of a software system is an *internal* quality.
- Maintenance consists of *adaptive*, *perfective* and *corrective* maintenance [IEEE 92]. These activities are related to portability, modifiability and testability respectively. Understandability supports all maintenance categories.
- The expenditure of maintenance consists of four variables: time, resources, correctness and quality.
- Documentation has a significant role when it comes to the maintainability of a system. It serves as a communication vehicle and comprises knowledge about the behavior and internal structure of a software system. Enhancing documentation can reduce the effects of software aging.
- The applicability of documentation depends on information needs of stakeholders and how information is presented.
- Documentation should be maintained, and therefore maintainable.
- Not all relevant architectural knowledge about the system can be retrieved.

Chapter 3

Problem Analysis

3.1 Introduction

In order to find a suitable solution for the maintenance problem regarding the LBCMS at Lost Boys, it is imperative to understand the cause and its effects which contribute to the problem. Without understanding the problem deeply and broadly, it is more likely that the proposed solution will be wrong. Also, it is important to realize that a problem usually has more than one cause. A cause can have multiple effects which contribute to the problem. It is considered a good practice to decompose a problem into more comprehensible subproblems, which can be considered in isolation [Jackson 05]. The aim of this research is not to deal with all subproblems, because this is nearly impossible to achieve within a timeframe of three months. The aim is to document the software architecture and requirements of the system, and understand how software architecture documentation can contribute in eliminating a part of the problem. In this chapter, the problem is analyzed and a hypothesis is formulated which encompasses the proposed solution.

Approach

The problem analysis is carried out as follows:

- The problem is described according to the perception of the problem by Lost Boys employees (the stakeholders).
- The causes and effects of the problem are presented by using an Ishikawa diagram.
- The problem is analyzed by asking “why”, it is important to know why Lost Boys thinks the problem is actually a problem. This leads to the problem behind the problem.
- The stakeholders who raise the problem are described by means of personas. Personas are fictitious characters which represent stakeholders within a targeted demographic. The personas are used in Chapter 7 to validate the solutions.
- The impact of the problem is described in order to determine the necessity of finding possible solutions to the causes of the problem.

Finally, the proposed solution of enhancing the documentation is assessed against the problem. It becomes clear why and how documentation can contribute in solving a certain part of the problem. This is formulated in a hypothesis, which will be answered in the remainder of this thesis.

3.2 The problem description according to Lost Boys

Maintenance of the LBCMS is a difficult, time-consuming, and consequently an expensive undertaking. Lost Boys' stakeholders think the costs of maintenance can be reduced. The following subproblems are identified by the stakeholders who contribute to the maintenance problem.

Knowledge of the LBCMS is tacit: employees at Lost Boys express concerns regarding their knowledge of the LBCMS; the organization depends heavily on oral communication and tacit knowledge of only one developer.

Understandability of the software: developers find it difficult to comprehend the LBCMS. Also, the process of deploying the software to hardware components is unclear. Recall from Chapter 2 that these factors contribute to the deterioration of software. This is a symptom of an underlying problem; the perceived complexity of the software and the lack of an abstract reference model.

The applicability of the system is unclear, because technical consultants do not know what features the system has and therefore do not include the system when they advise customers about the implementation of a content management system for websites.

In addition, there are no resources available at Lost Boys to deal with the problems mentioned above.

3.3 The causes & effects

The Cause and Effect Analysis is a diagrammatic technique often used to explore and display a problem, the cause of the problem, and the subsequent effects. This technique is valuable for this research, because it illustrates the causes of the problem. The cause and the subsequent effects keep the problem in place. The diagram provides an indication where to look in order to deal with (a part of) the problem. The following sections clarify the causes and effects and ultimately a choice is made between possible solutions to deal with the (sub) problem.

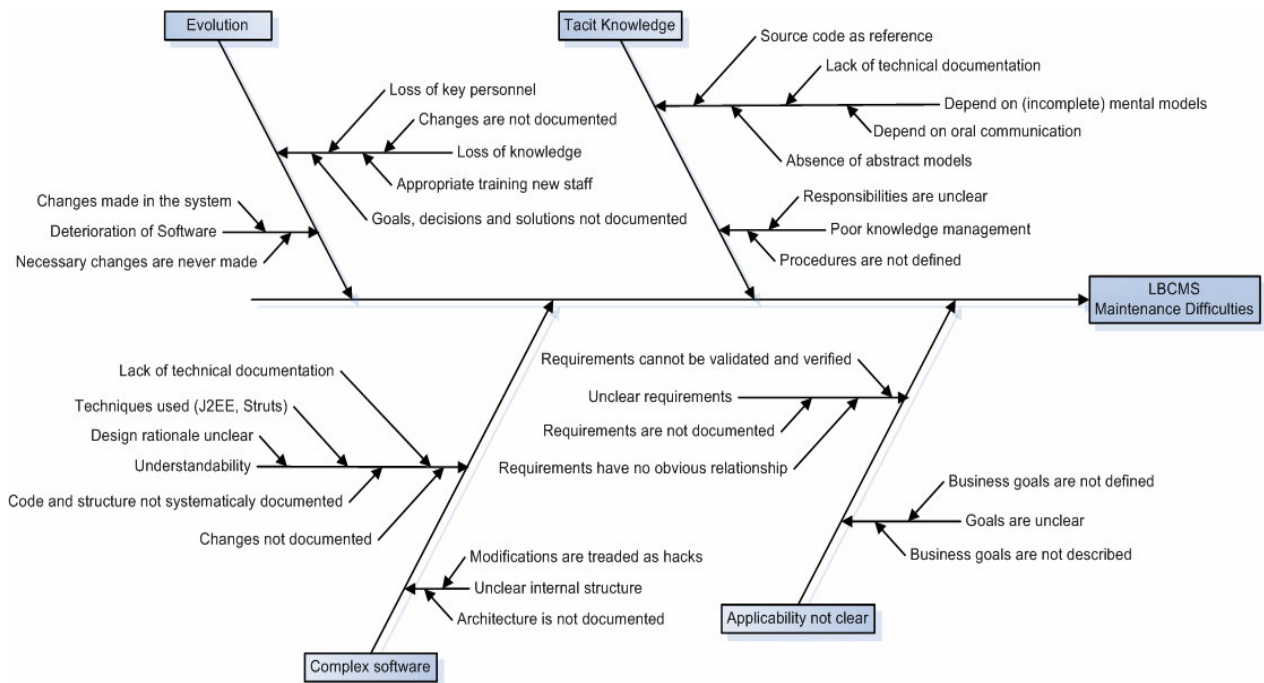


Figure 6 Cause and effect analysis

3.4 Why is this a problem

The Content Management System of Lost Boys was built three years ago by a small company named Inpact. After the implementation of the system, Lost Boys acquired the company and the software system. Currently, the LBCMS is in the evolution stage of its life-cycle. This means that the system is valuable for the customer as well as for Lost Boys [Bennett 99]. It also means that vast resources are spend to maintain the system [McConnell 04]. Lost Boys typically provides maintenance as a service for which monthly fees are charged. When a change is to be made in the system, it is desired to perform these changes as fast and accurate as possible. Otherwise, the costs of change will exceed the income from monthly fees. If nothing is done to enhance the support for quality attributes such as maintainability, the software quickly enters the servicing phase and eventually will be phased-out [Rajlech 01, Parnas 01].

Knowledge of the LBCMS is lost during the takeover of Inpact by Lost Boys. Key personnel left the organization and documentation is almost non-existent or written from the perspective of only one stakeholder or concern. There are no well-defined procedures for knowledge management, so nobody knows exactly what to do and when to do it when it comes to making knowledge explicit and transferable. Furthermore, it is unclear what design decisions were taken during initial development and what goals the stakeholders have with the system. Goals and design decisions are considered fundamental knowledge which shape the architecture of the system [Bass 04].

The deterioration of the software is a symptom of a problem which concerns the understandability of the code and structure of the system. Code is perceived complex because a lot of technical frameworks are used, some classes have a fairly large amount of lines of code, code is poorly documented, and the cohesion of classes is low (classes have more than one responsibility, which is considered bad design [Fowler 99, McCabe 89]). Figure 7 presents a computer generated view of the software modules in the LBCMS. The figure exemplifies the internal complexity of the LBCMS with which a software engineer is confronted when browsing the source code. Understanding a program by navigating through the code remains a difficult task for a programmer. The information is too detailed and the design rationale is missing; it is difficult for a programmer to develop an accurate abstract (mental) model of the system.

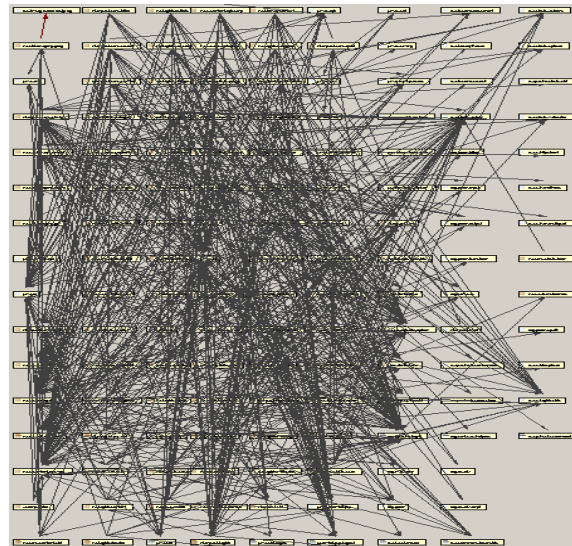


Figure 7 a computer generated module view of the LBCMS (Eclipse Inspector Gadgets)

The applicability of the system is not clear for technical consultants. In order to make a profit with the LBCMS, it is imperative to implement the system for at least 5 to 10 customers. At the moment, technical consultants do not include the LBCMS in their advice to customers regarding the implementation of a CMS like system. Consequently, the revenues of the LBCMS are low.

3.5 The problem from the perspective of John Foo; a Software Engineer

Consider John Foo as a Java programmer who has been working as a maintenance engineer at Lost Boys. After the takeover of Inpact, John has new colleagues and is now partially responsible for maintaining the LBCMS. John is asked to add a new feature upon request of the customer. At this moment the need for knowledge increases, because John wants to develop the desired feature and safely release an updated version of the LBCMS. What does John do to gain knowledge and perform his task? What problems does he encounter?

Obtain sourcecode and a working instance

First off all, he will need to obtain the latest sourcecode version of the LBCMS in order to set up his development environment and compile and run a fully functional instance of the LBCMS for testing and deployment purposes. The LBCMS is a system which is build in the Java J2EE language

and uses several other frameworks and techniques. In order to perform his tasks, John needs to install these technologies in his development environment.

- John needs to know exactly which techniques are used and where to find them. He will find that this information is not explicitly available. Consequently, much time is spent when a maintenance engineer needs to set up his workplace.

Understanding the software and its context

In order to understand the structure and the behavior of the LBCMS (software and hardware), John creates a mental model of the system; he asks his new colleagues for assistance, he searches the corporate network for documentation, he browses through the source code and he studies the features and behavior of the system. This process imposes several problems for John.

- His new colleagues know the system fairly well, however John does not know who has the knowledge he is looking for. In addition, when knowledge is communicated by people, they can only share information of which they know about. And even then, a lot of information remains unspoken (tacit knowledge).
- By browsing the sourcecode and analyze the features of the system, John develops a mental model of the software. He uses this model as a reference in order to perform his maintenance tasks and to communicate about the software with others. Sourcecode is a result of design decisions which were made and implemented during the development of the system. Researchers believe that it is impossible to comprehend a system by looking at the sourcecode only [McCabe 76, Boehm 83]. The absence of a formal abstract model of the software, such as documented software architecture, is a problem.

Change the software

With the knowledge John has gained, he now wants to make the necessary changes to the software in order to implement the new feature. This involves writing new code, modifying existing code, verifying if the changed software does not have any defects or cause failures, and finally John builds and deploys the new version of the software on the hardware. The problems John encounters during this process are:

- Code standards are not defined which leads to different styles of programming, which consequently leads to less comprehensible code [McConnell 04].
- Code is rarely documented; this makes algorithms and functions obfuscate, especially when a function has more than one responsibility (low cohesion) [McCabe 76].
- Test procedures and tools are not defined; this makes it hard to assess the correctness of a change.
- The hardware configuration is not documented; how does John know what software is deployed on the hardware?
- Should documentation be updated?

3.6 The problem from the perspective of Miranda Bar; a Technical Consultant

Besides John Foo, other stakeholders experience problems. Consider the situation of Miranda Bar, a technical consultant at Lost Boys who is responsible for advising customers about a CMS solution. In addition, Miranda is a decision maker regarding the LBCMS. When a change is needed, she is consulted regarding the approach to implement the change. Also, she assesses the costs of a change and decides if the change should take place (cost/benefit).

Understanding the capabilities of the LBCMS

Miranda needs information about the capabilities of the LBCMS when advising a customer. When a customer decides to implement a CMS-like solution, Miranda produces a “short-list” in which (3rd party) CMS systems are selected that suit the requirements of the customer. Based on the comparison of the capabilities and qualities of the systems (often referred to as *features*) on the short-list, a suitable solution is selected and implemented. When information regarding the capabilities of the LBCMS is not available, their advice might be wrong. Consequently, Miranda discards the LBCMS.

Making decisions about the LBCMS

Miranda is often consulted by software engineers, like John Foo, when changes need to be made to the LBCMS. Miranda has extensive knowledge about technology and uses pragmatic approaches to implement various solutions to a certain design problem or organizational issue. However, she needs information about the LBCMS concerning business goals, features, and quality requirements in order to make the *right* decision. She requires information to manage changes when the LBCMS is modified to align with business goals and processes of a customer. For example, when requirements are not clear, it is difficult to assess whether a new requirement introduces conflicts with existing requirements. In addition, she will require a “big-picture” overview of the system in order to determine where the change should occur and how the change affects the internal structure of the system, while preserving the conceptual integrity of the LBCMS.

Asses the cost of changes

It is necessary to determine what and how much resources are needed when a change is pending, in order to assess the cost of change. Miranda needs to know what features are already implemented in the system in order to assess whether the modifications will affect the features of the system. She also needs information about the internal structure of the system to determine where the modifications should occur. She requires tracing the goals and features to the architecture and eventually to the implementation, in order to estimate how much resources are needed to implement required changes.

3.7 The impact of the problem

The problem has a relative big impact regarding the maintenance of the LBCMS. Due to the lack of knowledge (such as design rationale) and the perceived complexity of the software, maintenance is an increasingly difficult and valuable undertaking. Based on the personas John Foo and Miranda Bar, the impact of the problem is:

- Time is lost when maintainers need to obtain the latest version of the source code of the LBCMS and install a working instance on their development PC's.
- Creating a mental model of the software is difficult, since source code is the only reliable architectural evidence and design rationale is missing. It is difficult to assess whether a change will improve or decline the quality of the LBCMS.
- The lack of a big-picture overview makes it hard to determine where changes should occur. Consequently most changes are made in “familiar-area's”, which resulted in classes with over 4400 lines of code and multiple responsibilities which is considered bad design [Dudney 02] and makes maintenance even harder.
- Ad-hoc modifications (hacks) increase the deterioration of the software [McConnell 04]. John experiences problems with verifying the correctness of changes and possible errors are not detected. Eventually Miranda and John (or the customer) will decide that the software is no longer viable for business and the LBCMS will be phased-out.
- Technical consultants, like Miranda, are not familiar with the capabilities of the LBCMS. As a result, the LBCMS is not implemented for new customers. The cost of maintenance increases, while no new income is produced.

It becomes clear that it is viable to find a solution for the problems of the personas John and Miranda. However, the extra time and resources needed to maintain the system are well compensated by the customer. This being said, dealing with the problems can create higher revenues for Lost Boys or lower prices for the customer. Lost Boys should assess which costs can be reduced in order to achieve this.

3.8 Possible solutions

There are several possible solutions to deal with the maintenance problems; the causes can be eliminated, the consequences can be reduced, the situation can be changed, the problem can be accepted, the forces that keep the problem in place can be reduced, or the forces that reduce the problem can be enforced. Either way, the solution must be suitable and feasible to implement at Lost Boys within the timeline of three months. In chapter 2 the characteristics of software evolution, maintenance and the role of documentation during the maintenance phase of the software life-cycle are discussed. Based on chapter 2 and the problem analysis, a list of possible solutions is presented below, which fall in the category of eliminating the causes and enforcing the forces that reduce the problem;

- Rewrite parts of the software which are complex (research should be done to define what exactly is meant by complex software and determine which parts fall in this category).
- Freeze requirements in order to prevent the implementation of new requirements. More time can be spend on fixing errors.
- Enhance the comments in the code so programmers have a more clear understanding of what the code does.

- Increase the price of maintenance for customers; generate more income in order to compensate the increasing costs.
- Converge towards system close-down, perhaps the software is at its end of its life-cycle and the system could be replaced by a new system.
- Enhance the software documentation, in order to capture relevant knowledge and make this knowledge transferable.

At the beginning of this research it became clear that enhancing the documentation would be most desirable and feasible. The following software documentation solutions were under consideration:

- Software Requirements Specification
- Software Architecture Documentation
- Technical Documentation

Software Requirements Specification

A Software Requirements Specification (SRS) comprises documented business goals, stakeholders involved, project drivers, functional- and quality requirements, issues and risks [Lauesen 02]. An SRS can be used to deal with the following subproblems; 1) understanding the context and behavior of the system; 2) identify dependencies between features, 3) define important quality requirements. The aim of developing such documentation is to provide information for Miranda Bar in order to advise customers about the LBCMS, support the decision making process, and assess the impact of change. The capturing and analysis of the requirements is discussed in Chapter 5.

Software Architecture Documentation

Software Architecture documentation provides an overview of the internal structure of software. It includes relationships with the environment (context) of the system and construction principles to be used in design of software components [Bass 03, Kruchten 95]. The architectural design is shaped by decisions about the design and the quality requirements. The design decisions are partially based on business goals and the requirements comprised in the SRS. A requirements specification and an architecture document are closely related to each other.

The aim of retrieving and documenting the architecture of the LBCMS is to provide a transferable and abstract blueprint of the system, accompanied by design rationale. This should enable stakeholders to develop a clearer model of the system and consequently the communication about the system is more effective and accurate. In Chapter 6, software architecture for maintainability is defined and the capturing and analysis of the architecture is discussed.

Technical Documentation

Technical documentation includes documentation of code in code, algorithms, interfaces, and APIs (Application Programming Interface). The goal of such documentation is to clarify difficult parts of the software on a relative low level, to assist a programmer whilst browsing through code to build the

mental model. Like all other forms of documentation, it is necessary to determine what is relevant to document to prevent that the documentation obfuscates the interpretation of the reader. A guideline could be developed which specifies when and what to document.

3.9 Hypothesis

The lack of documentation of the architecture and requirements is a problem at Lost Boys with respect to maintenance of the LBCMS. This research focuses on capturing and documenting the software architecture of the LBCMS. The problem analysis resulted in the following hypothesis:

The Software Architecture Documentation is complete enough when:

- 1) It provides explicit and transferable information about features, quality requirements, design decisions, and architectural views;*
- 2) It visualizes relevant views on the system to support the maintenance engineer' perceptual ability for interpreting complex information about the LBCMS;*
- 3) It enables abstract reasoning about the architecture by stakeholders, in order to identify issues, make rational decisions, and locate where changes should occur.*

How is this achieved? The remainder of the thesis answers this question.

Chapter 4

Approach for retrieving architectural knowledge

Software maintenance and the lack of explicit knowledge is an important motivation to capture software architecture knowledge. The problem analysis describes several concerns related to the maintainability of the LBCMS. In order to answer the research questions a hypothesis is developed. In this chapter, the approach for developing the software architecture documentation is described.

4.1 Existing methods for retrieving architectural knowledge

Many approaches for retrieving and reconstructing the architecture of a system are available. The goal of retrieving valuable information is “analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information” [Muller 00]. In [Tilley 00], three categories of techniques are defined to achieve this goal; browsing through source code, leveraging corporate knowledge and experience, and computer-aided techniques.

During the evolution of software, modifications and enhancements are often poorly documented, in these cases source code is believed as the only reliable source of information [Muller 00]. Consequently, most methods for retrieving knowledge concentrate on reverse engineering of source code. Examples are; design pattern recognition [Rich 90], call graph visualization, complexity measure [McCabe 76], and structural or behavioral visualization [Pauw 98].

However, information captured by the techniques mentioned above does not provide knowledge about business goals, design decisions, and requirements. A method to obtain this information is proposed by [Stoermer 03]. His method encompasses four activities; 1) extract information from available documentation, 2) interview experts, 3) organize a workshop with stakeholders in which the architecture is discussed, and 4) use available sources to retrieve information, such as source code. Other approaches mostly have similar techniques to retrieve knowledge [Bass 03, Muller 00]. According to Bass et al. (2003) and Lauesen (2002) the following activities are involved in creating software documentation:

- Creating the business case for the system (LBCMS vision and context)
- Understanding requirements (stakeholders and concerns, business goals, requirements)
- Creating or selecting the architecture (from the perspective of a related set of concerns)
- Documenting and communicating the architecture (design decisions and views)
- Analyzing or evaluating the architecture (validation of the results)
- Ensuring that the architecture matches with the implementation

4.2 Approach

The approach for the remainder of the project is illustrated in Figure 8. The approach comprises activities and techniques proposed by literature, which contribute to the confirmation of the hypothesis.

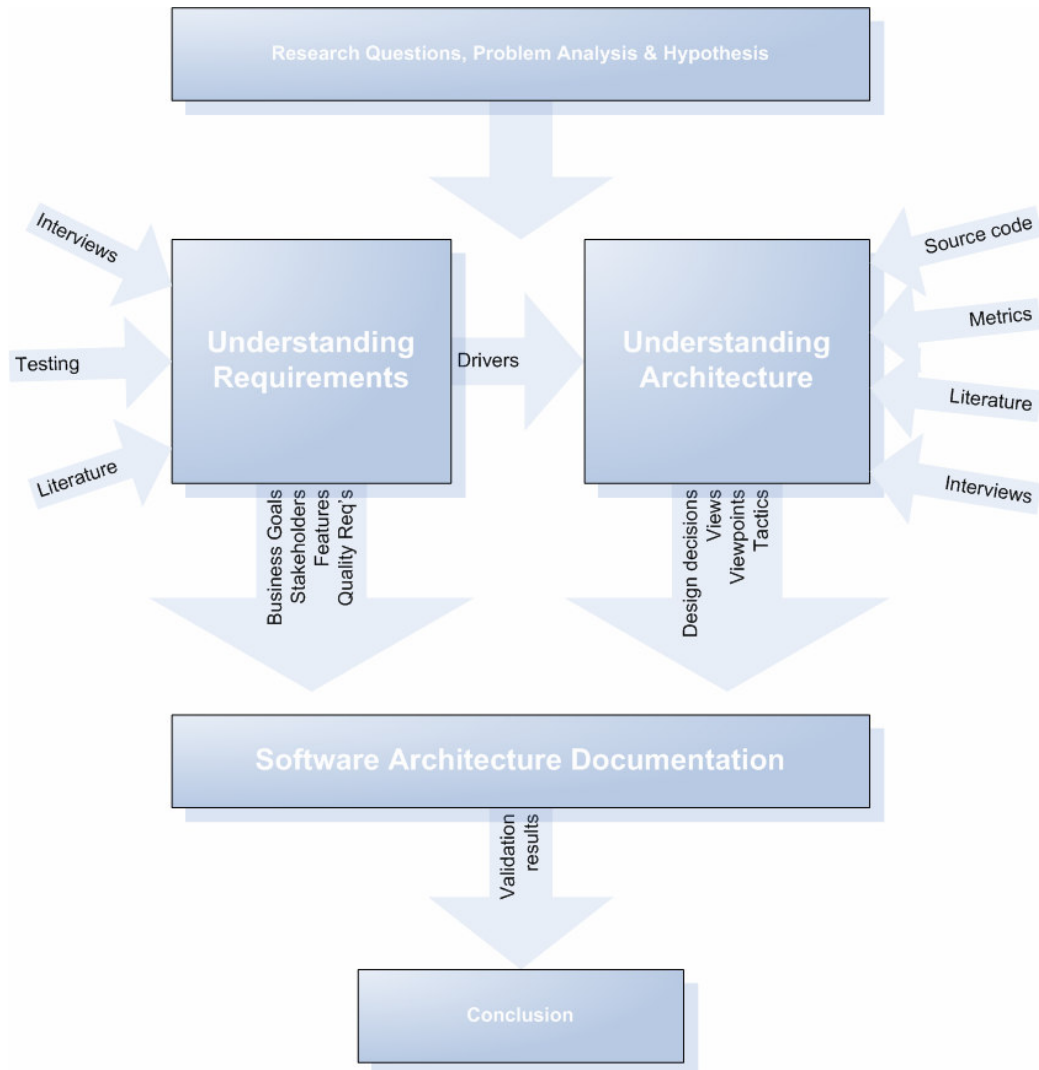


Figure 8 approach for retrieving relevant knowledge

4.3 Understanding the Requirements of the LBCMS

The aim of understanding requirements is to understand what to build *before* it is build, so the expensive development of a system is more accurate, verifiable, and reliable. This implies that requirements analysis should be done before the maintenance phase starts. In this research, an effort

has been made to understand requirements *after* the system was built. Many approaches to analyze upfront requirements are proposed by literature [Lauesen 02, Wiegers 04]. But what is applicable for a system already built? Based on a literature survey, the following recognized aspects of requirements engineering are selected to develop documentation for maintenance:

- **Stakeholders**: stakeholders are individuals or organizations who stand to gain or lose from the success or failure of a system [Nuseibeh 00]. The starting point of retrieving and documenting the requirements is to identify important stakeholders and capture their major concerns with the LBCMS.
- **Business Goals**: a business goal is a high-level objective of an organization or customer which is satisfied by the system [Wiegers 04]. Documenting the business goals is imperative to understand why the system is build. The documentation developed in this research encompasses business goals of a (typical) customer, but also the goals Lost Boys has with the system.
- **Features**: a feature is a set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business goal [Wiegers 04]. Features are documented primary for Miranda Bar; the capabilities of the system become clear and the dependencies between features and business goals enables Miranda to assess the impact of changes.
- **Product Vision**: a vision on the product is a long-term strategic objective of the purposes and form of the system [Wiegers 04]. The vision scopes the product towards an ultimate goal, as a result the future changes need to fit in the vision (or maybe the vision should be reconsidered). Understanding the vision helps maintainers to understand why certain changes are desired.
- **Quality requirements**: quality requirements basically define *how good* certain important aspects of a system should perform [Lauesen 02]. In this research, the maintainability of the LBCMS is the starting point of eliciting quality requirements. The requirements need to be specified in a consistent manner, and be measurable.

The capturing of information is done by analyzing the functions of the system and interviewing stakeholders. Interviews are used during the requirements engineering phase in order to identify the stakeholders and their stakes, the business goals and quality requirements of the system [Lauesen 02].

In order to verify requirements and to support requirements management, it is essential to uniquely identify requirements and relate requirements to business goals and vice versa (tracing) [Lauesen 02, Wiegers 04]. The principles and techniques are used during this phase, which consisted of three steps; capture, analyze, and document. This resulted in the *Software Requirements Specification*, which is merged with the architecture documentation at the end of this project.

4.4 Understanding the Software Architecture of the LBCMS

The architecture of the LBCMS was never captured in documentation. In order to understand the architecture and manage the evolution of the LBCMS, it is essential to document the architecture [Bass 03, Bennett 00]. How can this be done and what should be documented? The following activities were selected based on recommendations found in literature.

- Recovering architectural evidence; Bass et al. (2003) suggest the use of tools to accomplish this. In this project, several programs were used to extract factual data from the system. A widely accepted software metric, McCabe’s Cyclomatic Complexity Number or McCabe index, was used in order to understand the architecture of the system. The McCabe index determines the number of independent paths through a program, which is thought of as an indirect measure of maintainability [McCabe 76].
- Retrieve the three most important design decisions; during the design of a system, decisions are made which have downstream implications (rules, guidelines, or constraints) on the software [Bass 03]. Documenting these *design decisions* supports the maintainability of the system, because design decisions provide a rationale for the current architecture, and makes suggestions about avoided alternatives [Kruchten 04].
- Select and document architectural views; an architectural view is an abstraction of (a part of) a system from a particular perspective. A view addresses relevant concerns, while excluding elements that are not relevant to its perspective. A well known approach for developing views is the 4+1 View Model of Software Architecture, which is grounded on 4 views, with scenarios (hence the +1) to relate the views to each other [Kruchten 95]. After careful consideration, the decision was made not to follow the 4+1 approach due to time constraints. Instead, three views were developed based on relevant concerns of the stakeholders.
- Develop a “architectural strategy” for maintainability; when a system is designed, architectural strategies shape the architecture of a software system. To achieve quality requirements, decisions are made about the usage of a combination of these *tactics* [Bass04]. Tactics have a strong relationship with design decisions and software quality. A strategy for maintainability is developed based on the requirements, which incorporates tactics for modifiability, testability and portability. The strategy is verified with the tactics currently used in order to discover deficiencies and point out opportunities.

The activities are executed during the design, analyze, and document steps of the Understanding the Software Architecture phase. Finally, the requirements documentation (SRS) and the architecture documentation are merged in order to obtain one document which comprises the necessary information for John Foo and Miranda Bar (discussed in Chapter 3). The result is the Software Architecture Documentation (Appendix A).

4.5 Validation of the results

During this research it became clear that the success of the documentation depends on the correctness, completeness, and applicability of the documentation. Three tests are conducted in order to validate the completeness and correctness of the documentation;

- Testing feature descriptions by verifying the descriptions with the actual features of the LBCMS.

- A review workshop in which stakeholders validate the business goals, stakeholder descriptions and quality scenarios.
- Validating the metrics by comparing results of multiple tools.

The applicability of the documentation is validated by:

- A review workshop in which fictional maintenance scenarios are used to assess how and when the documentation is useful.
- A new architectural strategy for maintainability is developed based on the tactics presented in the documentation. The tactics depicted in this strategy are compared with the tactics used in the LBCMS. The documented strategy should provide a platform which guides future improvements of the maintainability of the LBCMS.
- Reasoning about the applicability of the documentation for the personas John Foo and Miranda Bar.

Chapter 5

Understanding Requirements: Software Requirements Specification

Requirements engineering is the process of gathering requirements and weighting requirements against each other and against the business goals [Lauesen 02]. The main goal of specifying the requirements was to support perfective maintenance and requirements management. Perfective maintenance and requirements management benefit from the description of the *stakeholders*, the *business goals*, the *features* supported, and the *quality requirements* [Bennett 00, Kruchten 04]. The requirements specified for the LBCMS are based on empirical research by the author, resulting in a Software Requirements Specification (SRS, see Appendix A). This chapter describes the process of capturing, documenting and analyzing requirements.

5.1 Approach to Requirements Engineering

Figure 9 illustrates the approach to *requirements engineering*.

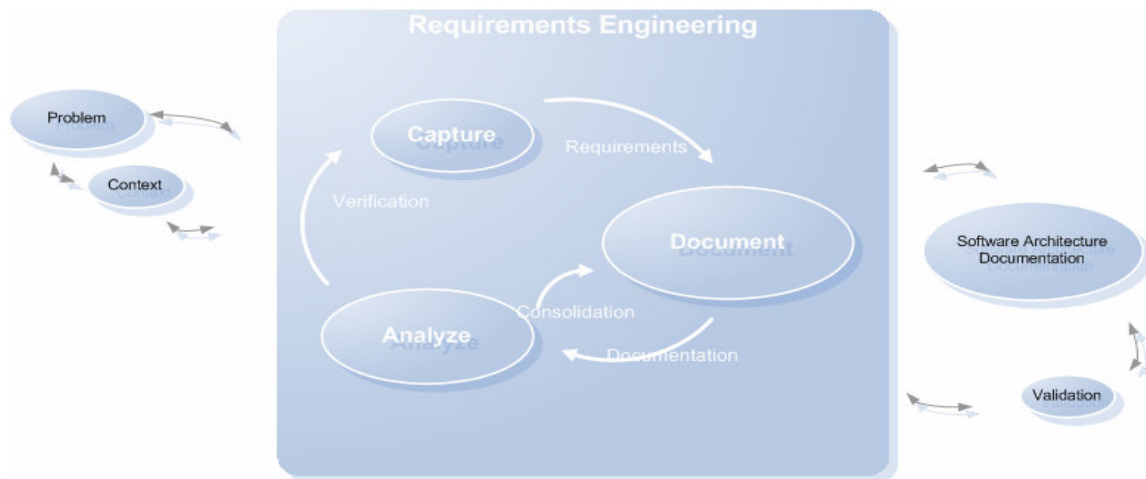


Figure 9 approach to requirements engineering

Three activities are identified in the figure above, which form the basis for the execution of the requirements engineering phase;

- Capture is the process of retrieving goals and requirements; interviews were held and the functionality of the LBCMS was tested
- Document is the process of documenting the goals and requirements.

- Analysis is the process of tracing, verifying and validation of the requirements against stakes, business goals and the current implementation of the system.

5.2 Stakeholders and their Concerns

The first step taken in the capture process was the identification of the stakeholders. Stakeholders are individuals or organizations who stand to gain or lose from the success or failure of a system [Nuseibeh 00]. In order to achieve the goals of this project and develop pragmatic documentation, it is essential to take the concerns of the stakeholders into account.

Capture Stakeholders and their Concerns

Only stakeholders from within the host organization were interviewed, because the research focused on *internal* quality attributes of the LBCMS (maintainability, modifiability, testability, understandability). The requirements of the customer and users were based on assumptions and the observable behavior of the system. During the interviews, several questions were asked in order to identify:

- The stakeholders of the system
- The goals the stakeholders see for the system
- Their involvement and concerns
- The opinions about the system.

Precisely capturing all stakeholders and their concerns was for many reasons a difficult process; time constraints, the absence of knowledge about the LBCMS, and the unavailability/unwillingness of key personnel required a vigorous approach. The concerns of stakeholders were limited to a maximum of three. This was sufficient, because the context of the problem provided enough concerns to relate the findings to literature and ultimately chose the right architectural views.

Documenting Stakeholders and their Concerns

The following stakeholders with their stakes in the system and project were captured and documented as follows:

Table 1 Captured stakeholders

Role	Stake
Technical consultant	Needs information about features and architecture. Advises (potential) customers about content management systems. Discovers technical opportunities.
Architect – Back-end developer	Needs specifications, abstract design, code conventions, documentation. Negotiates and makes tradeoffs among requirements. Desires less complexity and less different techniques.

Functional Maintenance	Needs information about the operational state of the software. Performs maintenance and provides support to users.
Project Manager	Needs to estimate development efforts, allocates resources, plans development and deployment, controls project execution and reports progress.
Front-end developer	Develops the Front-end (GUI) of the system. Needs information about style guides, features and quality attributes.
Application Management	Needs information for deployment of the system on the technical infrastructure.
Distributor (Publisher)	Needs to put content on the distributor website. Communicates with (potential) customers. Desires to monitor content demand and popularity.
Dealer (Publisher)	Needs a showroom on the internet, needs distributor information concerning marketing.
Mitsubishi Motors	Needs uniformity, establishes corporate identity, measures popularity and needs sound communication. Desires faster updates and adequate estimates concerning changes.

Analysis of Stakeholders and their Concerns

The stakeholders expressed several concerns they had with the system. Technical consultants need information about the capabilities of the LBCMS in order to include the LBCMS in their advice to customers. Developers need technical information, abstraction from code and components, and desire less complicated code. The conclusion based on the analysis was that understanding the system technically and functionally and less complexity was desired by all stakeholders.

5.3 Business Goals

Business goals help to create a vision of the high level organizational goals the system should support. Business goals basically answer the question why the system is built and maintained. The business goals needed to be documented because they are an important driver for the architectural solution and the actual implementation of that architecture [Bass 03]. In fact, the business goals form a foundation of the quality of the system, because the architectural solution (the tactics used to achieve qualities) must fit the goals stakeholders have with the system.

Capturing Business Goals

Lost Boys has certain business goals with the LBCMS and so does the customer (in this particular case, Mitsubishi Motors Europe). However, these goals were never explicitly formulated. This made it difficult for developers to understand the system, and trace requirements to business goals and vice versa. This also made it difficult to capture the business goals, because the initial goals and their rationale were lost over time. Questions about the necessity of requirements could not be answered. The necessity of the current functions of the system was determined by customer demand only.

The business goals were captured using interviews, assumptions, and common sense. A distinction was made with respect to the goals of Lost Boys and the goals of the customer. In the SRS, a detailed description of the business goals is included.

Documenting Business Goals

Table 2 shows how the most important business goals of Lost Boys were documented. A business goal has a unique identifier to enable tracing and support requirements management. Each business goal has a name, and a brief explanation is given to clarify the business goal.

Table 2 Captured business goals Lost Boys

Id	Business Goal	Explanation
BGLost 1	Full-service organization	Lost Boys offers its customers a comprehensive all-inclusive solution to their marketing and internet needs, which includes concept design, visual design, interaction design, websites, hosting and maintenance. A CMS plays an important role in this strategy; it offers customers “out-of-the-box” functionality and because all customers use the same system, training and support can be standardized, faster and cheaper.
BGLost 2	Reduce required resources	The implementation of an in-house developed and maintained CMS should reduce required resources by: a) suppliers of content management systems, b) development of custom content management like solutions, c) standardized development, maintenance and application management
BGLost 3	Time to market	The pre-developed features of the CMS reduces the time needed to implement a website which requires content management ability. A short time to market is often important for customers and thus a valuable marketing argument
BGLost 4	Quality Assurance	Quality is an important aspect for products delivered by Lost Boys. The CMS should support certain quality attributes in order to provide this quality to all customers.
BGLost 5	Customer responsibility regarding content	Lost Boys should not need to assist customers with managing content for their websites. Lost Boys cannot be held responsible for content.
BGLost 6	Vendor lock-in	The implementation of the CMS makes it difficult for customers to switch to another vendor. Although this is a valid business goal, it is a bad argument from a sales perspective.

Analysis of Business Goals

The business goals “reduce required resources” and “quality assurance” partially relate to maintainability. Reduce expenses of maintenance through standardization is mentioned. Also, it is desired to support quality assurance. An issue with quality assurance is the large amount of time needed between the discovery of an error or failure and fixing its cause. The LBCMS is applied in projects for multiple customers to reduce expenses, introduce standards, and achieve a short time to market. This introduces problems concerning new customer requirements, because the implementation of such requirement means all customers that use the LBCMS are affected. Also, it was impossible to assess if such requirements fitted within the goals of the system, because until now, the goals were never specified.

5.4 Feature Requirements

Features are defined by [Lauesen 02, Wiegers 04] as domain-level requirements which outline one or more related functions, tasks involved, and related data. Basically, features describe the capabilities of a system. Features are often used by the marketing of commercial of the shelf (COTS) systems because of their descriptive nature. Lost Boys uses feature descriptions of competing content management systems to select a system that best suits the requirements of a customer. Documented features also contribute to perfective maintenance, because they enable reasoning by stakeholders about the feasibility and impact of changes.

Capturing Feature Requirements

The feature requirements were captured using interviews and by testing the observable behavior of the system. This was a straightforward process of analyzing functionality and verifying the findings in interviews with developers. The capturing of feature requirements resulted in the following list of LBCMS features:

Table 3 List of LBCMS features

Features	
Feature 1 Webcontent management	Feature 12 Insite edit
Feature 2 Localization	Feature 13 Content Search
Feature 3 Baked pages	Feature 14 Dealer Search
Feature 4 Fried pages	Feature 15 Fast search (xml snippets)
Feature 5 URL Canonicalization	Feature 16 Scheduler
Feature 6 Media library	Feature 17 User Management
Feature 7 Authentication	Feature 18 Archive
Feature 8 Authorization	Feature 19 Record Locking
Feature 9 Workflow	Feature 20 Content templates
Feature 10 Preview	Feature 21 Content types
Feature 11 WYSIWYG editor	Feature 22 Database Indexing

Documenting Feature Requirements

The features of the LBCMS are documented as follows:

Feature #nr Name (a unique id for the feature)	
Version	The current version of the feature description
Date	The date of the current feature description
Description	A one sentence description of the feature
Rationale	A justification for this feature
Source	Trace to Business Goals (reference to section 2.3 Business Goals)
Fit criterion	A measurement of the requirement to test if the feature matches the requirement
Current issues	Concerns about the feature (optional)
Conflicts	Other features that conflict with this feature (optional)
Dependencies	A list of other features that have a technical dependency on this one (optional)

Figure 10 Feature description

Recall from Chapter 4 that versioning and traceability (verification of requirements and business goals) support requirements management in the evolution phase [Lauesen 02, Wiegers 04]. The unique id, version, date, source (reference to related business goals), and dependency attributes of a feature are meant for requirements management. The description and rationale provide information about the tasks supported by the feature and why the feature is included in the LBCMS.

Analysis of Feature Requirements

All features could be related to one or more business goals. This suggests that all features are compulsory. It also became clear that some features have technical relationships with other features. This is useful information when the system is modified; if a feature with dependencies is modified, the depending features could be affected. With explicit dependencies, verification of these features is possible in order to maintain the systems behavior. Documented features are useful in order to:

- Make the LBCMS a better competitor in the “CMS selection process”
- Support perfective maintenance, thus the maintainability of the system
- Serve as input for the architecture; the mapping of features to software components.

5.5 Quality Requirements

In Chapter 2, software quality is discussed. Recall that quality requirements shape the architecture of a system and determine its quality from a related set of concerns. The bottom line of capturing, analyzing, and documenting quality requirements is to consider all thinkable quality factors and assess the degree of importance [Lauesen 02].

Capturing Quality Requirements

To capture the requirements, interviews were held and the *Service Level Agreement (SLA)*¹ was analyzed. It was difficult to capture the requirements, because most stakeholders had no opinions about the *internal* quality attributes of the system. Furthermore, most interviewees defined software quality as performance (throughput, capacity) and availability (uptime of the service). In their opinion, good quality was high performance and high availability. This was clearly reflected by the SLA, because only those qualities are defined in the document. An attempt was made to capture maintainability requirements from stakeholders. This proved to be a difficult undertaking; the stakeholders had difficulties in expressing measurable requirements. Also, all stakeholders felt it was not their responsibility to define those quality requirements. Unfortunately, it is still not clear who is responsible. Assumptions needed to be made and responsibility taken to specify maintainability requirements. To achieve this, maintainability is decomposed into modifiability, testability, and portability. *Scenarios* were used to define modifiability, testability, and portability requirements (recall from Chapter 2 that maintainability was decomposed into these attributes). Although this made specifying quality requirements more complex, the results are comprehensible and verifiable. The capturing resulted in quality requirements regarding (summarized, see Appendix A for more detail):

- Availability; mean time between failures, response to server unavailability, reporting of availability
- Performance; scalability of servers to deal with large amounts of requests
- Maintainability; requirements were specified for modifiability, testability and deployment, such as “2 to 3 components are affected when a functionality change is made” and “An automated test should have a branch coverage of at least 25% percent”

Documenting Quality Requirements

To understand the landscape of quality requirements of the LBCMS, the quality grid of McCall and Matsumoto (1980) was used. In this grid, important quality attributes are presented. Each quality attribute can be qualified. Concerns of requirements that need special attention need to be recorded. Figure 11 presents the quality grid of the SRS of the LBCMS.

¹ An SLA is a contract between the customer and Lost Boys regarding the operational state and maintenance of the LBCMS

Quality attributes	Critical	Important	As usual	Unimportant	Ignore
Operation					
Integrity			X		
Security			X		
Correctness			X		
Availability	1				
Usability			X		
Efficiency				2	
Performance		3			
Revision					
Maintainability	4				
Modifiability		5			
Testability		6			
Flexibility				7	
Transition					
Portability			X		
Interoperability			X		
Reusability					8
Installability		9			

Figure 11 Quality grid

Quality requirements need to be precise and measurable. It has to be clear what quality attribute is supported, how it is supported, and how to assess whether the quality is achieved by the system [Bass 04]. Scenarios were developed to specify critical and important quality requirements depicted in the quality grid. Quality scenarios were documented as illustrated in Figure 12.

QA 1	Unique name
Version	current version
Date	date recorded
Source	an entity that generated the stimulus
Stimulus	a condition that needs to be considered when it arrives the system
Artifact	(a part of) the system that is stimulated
Environment	the condition of the system
Response	what is done when the stimulus arrives the system
Response measure	a testable measurement of the response

Figure 12 Quality Requirement description

5.6 Conclusion

In this section, the findings so far are briefly summarized;

- Stakeholders have concerns which influence business goals, requirements and in the end the architecture of the system. By using the SRS, these influences can be understood.

- The business goals and features were documented for understanding goals and demonstrate the capability of the system.
- To support perfective maintenance, requirements should be manageable. This is achieved in the documentation through tracing between goals and capabilities, and versioning of all requirements.
- At Lost Boys, quality of the system was thought of as the availability and performance of the system.
- Maintainability requirements are difficult to specify, decomposing maintainability into modifiability, testability and portability make maintainability more comprehensible and quantifiable.
- Understanding stakes, goals and capabilities help stakeholders to understand the system, so reasoning about the system is supported by the SRS.

Chapter 6

Understanding Architecture: Software Architecture Documentation

With the development of the Software Requirements Specification, the necessary input for the *Software Architecture Documentation* is established. Software Architecture documentation is an abstract description of a software intensive system. It enables stakeholders to reason about the system and its quality attributes. The allocation of functionality on software components determines the architecture's support for quality. Bass et al. (2004) defined software architecture as follows:

“The structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationship among them.”

Boehm's (1983) definition of software architecture summarizes architecture as:

Software architecture = {Elements, Forms, Rationale/Constraints}

The aim of specifying the architecture is to enhance the quality of the documentation. Bass et al (1998) pointed out that there are three reasons why a software architecture specification is important:

- It serves as a communication vehicle for stakeholders and enables them develop a vision on the system; a model of the system enables stakeholder to reason about the system.
- It contains the first design decisions; these decisions had an significant impact on the design of the system
- It is a transferable abstract blueprint of a system; because it constitutes the system in a relatively small, transferable model.

Documenting architecture is hard. It is difficult to determine what makes the architecture “good” or when the documentation is complete. Architecture documentation is never complete, because it is impossible to cover all concerns and details. However, it still can capture enough knowledge which is relevant and useful for maintenance. Bass et al. (2004) propose the use of architectural views and tactics to address the communication needs from the perspective of related concerns. The architecture documentation should enable different stakeholders to reason about the architecture and identify problems and opportunities. This is achieved by developing views that address their concerns.

6.1 Approach

It seemed most suitable and feasible to analyze and document the concerns (communication needs served by the architecture), design decisions, and architectural viewpoints. Also, information of the system is extracted by using software metrics.

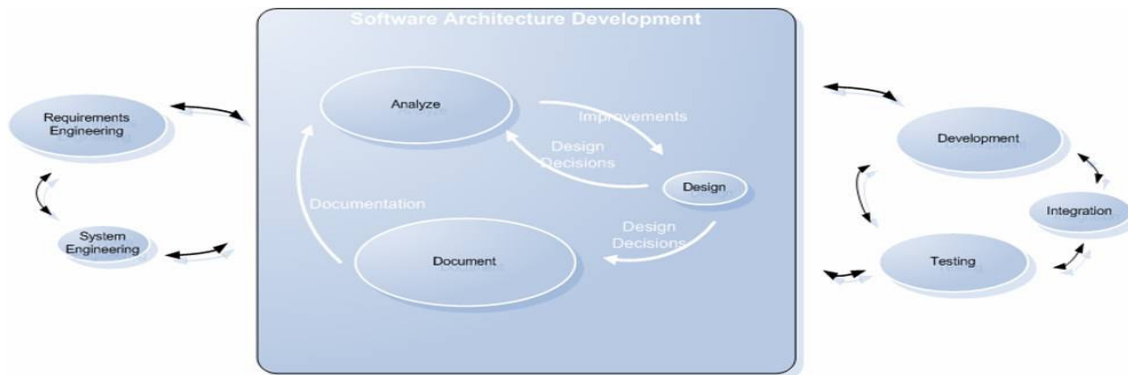


Figure 13 Software Architecture Development phase

6.2 Understanding the internal structure

As described earlier, browsing source code in order to fully comprehend a system is considered difficult and time consuming task. However, the changes made during maintenance are eventually made in the source code. Browsing source code provides understanding of the structure of the packages and code, the complexity, the techniques used, and the applied (or absence of) coding standards. During this research, a lot of time is used to gain this understanding by browsing through the source code.

Structure of the packages

When browsing through the package structure of the LBCMS, it becomes clear that a great effort has been made to establish a separation of concerns. The purpose of each package is revealed by its name and position within the package structure. By analyzing the packages, the purpose of a package and the implementation of certain techniques become clear. However, understanding the connections (communication) between the packages requires in depth analysis of the classes, descriptor files and routines. This makes the package abstraction less valuable for understanding the runtime behavior of system.

Browsing Source code

When browsing through the source code of the LBCMS, the application of object-oriented design and several frameworks becomes clear. Interfaces are used for the communication between packages and frameworks. This supports the flexibility of connections between major parts of the software (often referred to as loose-coupling). In order to keep the source code comprehensible, the classes and algorithms have understandable names and are frequently commented. Unfortunately, the absence of a guideline resulted in inconsequent names and a lack of comments throughout the software. This becomes clear when browsing through big classes (1000+ LOC) which are subject to frequent change. For example, the J2EE class `AdminAdministratorBean` has changed frequently when different builds in the code repository are analyzed. Another factor that makes the LBCMS difficult to

understand is that J2EE EJB uses metadescrptors (configuration files) to generate classes. To fully comprehend the EJB component of the LBCMS, it is necessary to browse through several xml based meta descriptors.

Software Metrics

In addition to browsing source code, software metrics were used to determine the size of the system (lines of code), the cohesion of classes (the number of responsibilities of a class), and the cyclomatic complexity (to determine the complexity of the system) [McCabe 78]. The Eclipse plug-in Metrics¹ was used to obtain the information. Two other tools were used to verify the results (see Chapter 7). Based on the results of the metrics used, the following conclusions are drawn:

- The packages within the CMS are tight coupled and have a great dependency on each other
- The CMS is a small to mid-size application with 70648 lines of code (LOC)
- A class has an average of 11.76 methods, which is acceptable [McConnell 04]. AdminAdministratorBean has 148 methods.
- Almost 50% of all methods have more than one responsibility, which is considered bad design (lack of cohesion) [Fowler 99].
- On average, a class has tight coupling with three other classes.
- On average, a class has 0.3 subclasses
- With a McCabe index of 17, the system is complex. McCabe (1978) argues that 10 is the threshold of understandable code.
- The AdminAdministratorBean EJB container has many responsibilities and dependencies, according to Dudney et al. (2002) this is a “God class” antipattern.

More results of this analysis can be found in the documentation (Appendix A). The facts obtained with the tool confirm that the system has complex parts. Most problems are found within the EJB component of the LBCMS. It becomes clear that engineers had little understanding of the architecture of the system and EJB in general; three classes are frequently changed without a clear reason why the change should occur in one of these classes. As a result, the classes have multiple responsibilities and are tight coupled with other parts of the system, which is considered bad design [Dudney 02]. In addition, the changed or new code is not consequent explained in comments, which makes the understandability of the internal structure even harder. Several opportunities can be identified in order to reduce the complexity of this component (discussed in Chapter 7).

¹ Eclipse is a Java Integrated Development Environment (IDE), Metrics is a plug-in for Eclipse, for more information see: <http://sourceforge.net/projects/metrics>

To visualize the internal complexity of the LBCMS, a component and connector view was generated with the tool Metrics, which is depicted in Figure 14. This figure was interesting because it illustrates the complexity and dependencies of the LBCMS. It is useless for maintenance, but useful to convince people of the necessity of reducing the complexity of the LBCMS.

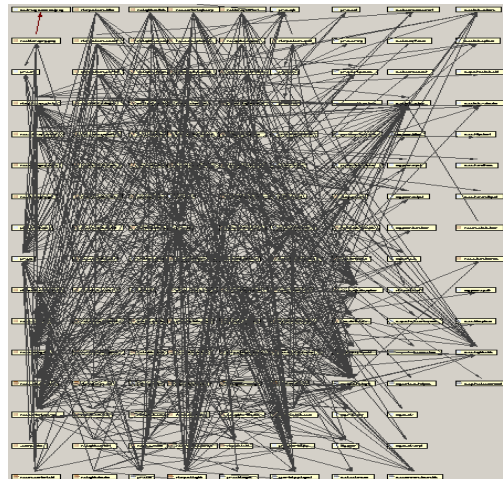


Figure 14 primary view of the metrics view

6.3 Design Decisions

Documenting design decisions is necessary in order to understand why certain decisions were made [Bass 04]. Decisions about the design of a system are primarily based on requirements, the environment, and the technical and domain knowledge of the development team. Uncertainties about the rationale of these decisions make maintenance more difficult, because the downstream effects of the decisions are not easily identified. Documenting the decisions enables reasoning about modifications in the architecture at later stages of the life-cycle, and enables assessments of the risks involved.

Documenting Design Decisions

Table 4 presents three of the ten identified design decisions (see appendix A for the complete list). The number indicates its influence on the system, so built for the future was identified as the most important design decision. Furthermore, a brief explanation of each decision is given.

Table 4 Major design decisions

Nr	Decision	Explanation
1.	Flexibility for future extensions	When the system was designed and built, flexibility for future changes was a major concern. To achieve this, it was believed that J2EE provided this flexibility.
2.	J2EE EJB	EJB is used to simplify certain design and development issues, such as separation of concerns, transaction management, data persistency
3.	Templates: XML, XSLT, JSP, and Taglibs.	Multiple techniques are used for merging templates with content for page creation.

Analysis of Design Decisions

During the development of the CMS, several decisions were made concerning the structure of the system. The CMS was initially built for Mitsubishi Motors Europe. Over time, other customers would use the system also. Decisions were made concerning the foreseeable future, the software and hardware configuration, the frameworks used and the implementation process. The most important decision was to make the system flexible for future changes. To achieve this, a separation of concerns was established (interface, logic, and data) and techniques were selected which were believed to support flexibility, such as Enterprise Java Beans and Struts.

6.4 Architectural Views

A *view* conveys a coherent set of architectural elements from the perspective of a related set of concerns. Views are probably the most important concept related with software architecture documentation [Bass 03]. Therefore, it was empirical to design views for the architecture documentation of the LBCMS.

Choosing Relevant Views

When specifying a view, several considerations must be made; who are the stakeholders and what are their concerns? If the concerns are identified, views that address these concerns can be designed. Clements et al. (2002) defined the communication needs of stakeholders which are served by architectural views. Recall that the stakeholders and their concerns were recorded during the requirement engineering phase, as described in Chapter 5. Table 5 outlines the communication needs of the stakeholders based on empirical research and [Clements 02], from the perspective of maintainability.

Table 5 Communication needs served by architecture

Stakeholders	Need information to
Architects and requirements engineers	negotiate and make tradeoffs among competing requirements, make transferable abstractions of the systems design
Developers	establish performance and availability, provide inviolable constraints on further developments
Maintenance engineers	reveal areas a prospective change will affect
Technical consultants	allocate and plan project resources, identify technical opportunities in terms of reuse, flexibility and portability
Quality Assurance engineers	asses whether the implementation is faithful to the architectural prescriptions

Source: Adapted from [Clements 02]

Four views were developed by the author based on the communication needs as depicted in Table 5. Table 6 describes these views and provides a brief rationale explaining why the view is important to include in the architecture.

Table 6 rationale for chosen views

View	Rationale
A Factual view which comprises the gathered facts about the systems size and complexity	This view does not address the communication needs of stakeholders directly; instead it provides useful information about the size and complexity of the system. This view can be used by technical consultants and developers to assess the viability of the system for business opportunities and identify issues.
A Component & Connector view in which the layers, the components, and the relationships among them are depicted	The view is developed for Maintenance engineers, Technical consultants and future architects. The quality attribute Maintainability was the driver for this view, therefore it shows what responsibility each component has, and how the components within the business logic layer communicates with each other and to other layers.
A Deployment and Allocation view that illustrates the hardware configuration and the allocation of software	The deployment and allocation view is useful for application management, maintenance engineers, technical consultants and software engineers, to understand the configuration of the hardware and the deployment and allocation of the software on the hardware (see Additional info). It also shows the redundancy which supports Availability and Performance.
A Module view that shows the necessary assets to develop templates	This view is developed for Front-end developers, maintenance engineers, customers and project managers to clarify the technique the CMS uses to assimilate and cache a webpage. It is important for these stakeholders to understand the principle of creating a webpage, because all of them use this technique.
A Decomposition view of the EJB component of the architecture	Based on the metrics described earlier, the EJB component of the system is subject to frequent changes during maintenance. Maintenance engineers need an overview of this component, its submodules, and Java Beans.

Documenting a View

Several approaches to documenting a view have been proposed by literature. In this research, a view has the following attributes:

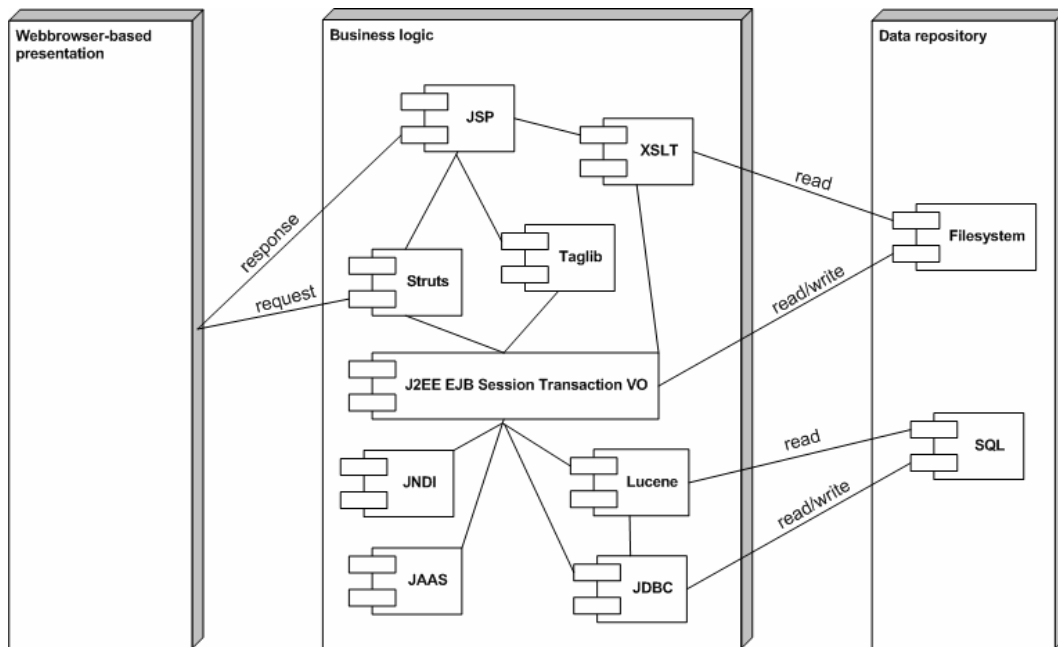
- Viewpoint name
- Stakeholders addressed
- Concerns addressed

- Variability points in the architecture
- Language and modeling techniques used

Based on the concerns and decisions described earlier, three views were developed; a component and connector view, a allocation view, and a runtime view (see Appendix A for more information).

Component and Connector view

The primary view of the Component and Connector view (Figure 15) presents the components and their relationships at runtime. The view is developed for Maintenance engineers (John Foo), Technical consultants (Miranda Bar) and future architects. The quality attribute Maintainability was the driver for this view, therefore it shows what responsibility each component has, and how the components within the business logic layer communicate with each other and to other layers. In addition, the viewpoint of the view comprises a list with the techniques and frameworks used within the system, accompanied with hyperlinks to relevant information on the internet about the item on the list. See Appendix A for the complete architectural view and viewpoint.



Key

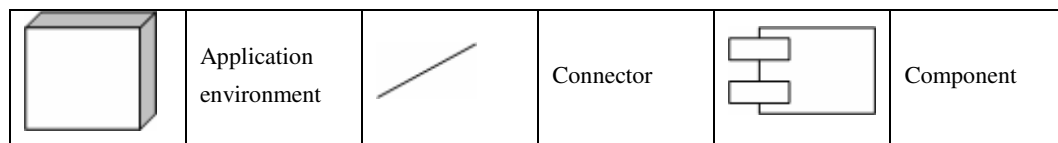
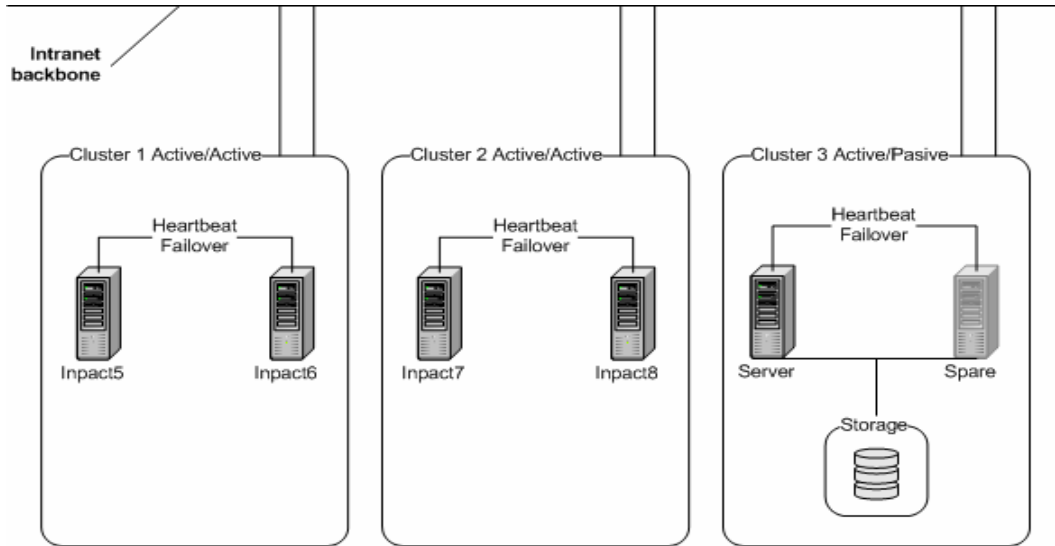


Figure 15 Component and Connector view

Allocation view

The primary view of the Allocation of the CMS software is useful for application management, maintenance engineers, technical consultants and software engineers, to understand the configuration of the hardware and the deployment and allocation of the software on the hardware (for additional info see Appendix A). It also shows the redundancy which supports Availability and Performance.



Key







	Server		Redundant server		Cluster of servers
	Ethernet crossover		Ethernet connector		Description
	Redundant shared data storage				

Figure 16 Allocation view

Runtime view

This view is developed for Front-end developers, maintenance engineers, customers and project managers to clarify the technique the CMS uses to assimilate and cache a webpage. It is important for these stakeholders to understand the principle of creating a webpage, because all websites managed by the LBCMS use this technique. The content templates in the system are used by end-users and are provided with small example images for understandability. See Appendix A for more information.

6.5 Maintainability Tactics

There is a connection between maintainability requirements and the architectural strategy. During this research, maintainability scenarios were difficult to specify. Unclear maintainability requirements (scenarios) are a problem which should be dealt with. This was achieved by decomposing maintainability into several quality attributes.

It is advised to implement a collection of tactics to achieve an architectural strategy for maintainability [Bass 04]. An architectural decision is made based on the quality requirements. This implies that choosing the right tactics is possible when all the requirements are known. This is not the case, because tactics can be specified that support an architectural strategy [Bass 04]. A tactic for maintainability was developed during this project. This tactic was based on the requirements described in the SRS. The tactic, presented in Figure 17, was used during the reasoning about the architecture to assess which tactics could enhance the systems maintainability. Basically, it summarizes the tactics that were used in this research to develop the architecture and validate it against the system. Also, the tactic is a product of the captured architectural knowledge. If the tactics are comprehensible and sufficient to reduce the maintenance problem, the architecture documentation is complete enough [Bass 04].

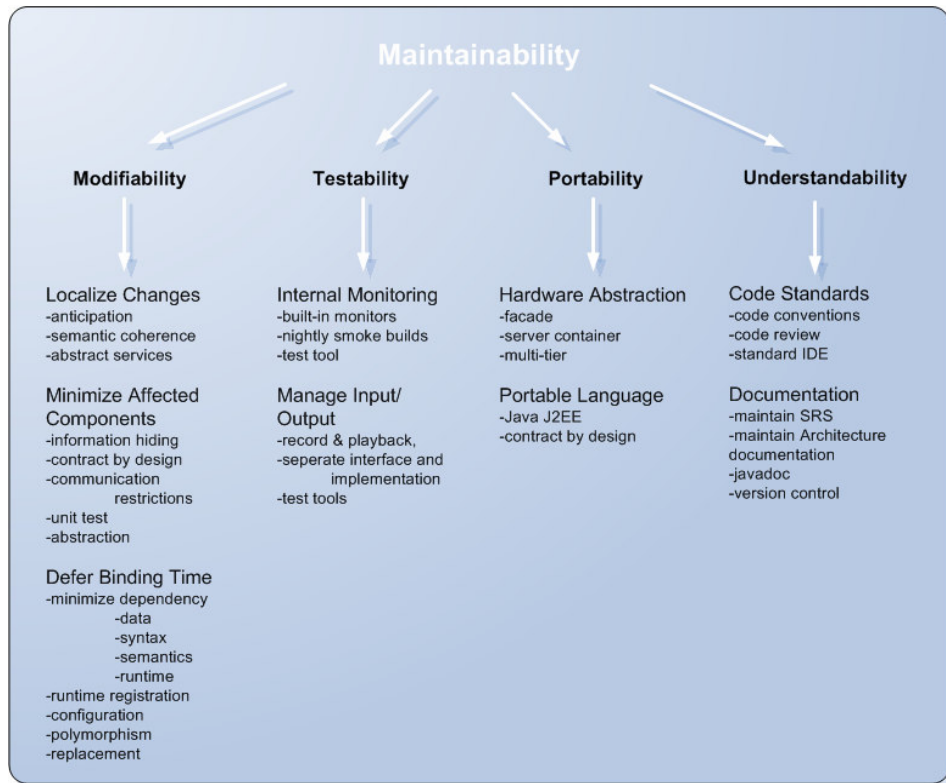


Figure 17 a Maintainability strategy with tactics

6.6 Conclusion

Architecture documentation is never complete, because it is impossible to cover all concerns and details. However, it still can capture enough knowledge which is relevant and useful for maintenance.

The architecture documentation includes a list with *design decisions*. The documented design decisions affect the maintainability of the LBCMS differently. Despite all proposed methods and techniques to retrieve design decisions, capturing all relevant design decisions, their rationale and the impact of these decisions proved to be impractical during this research. Therefore, the list with decisions is kept to ten, of which the top three are described in more detail (rationale, dependencies, consequences).

The quality attribute maintainability of the LBCMS was the starting point of documenting *views*. The views are developed from the perspective of stakeholder concerns and important qualities, including modifiability, testability, portability, and understandability of the system. The complexity view was developed to communicate facts obtained with software metrics; a model of the LBCMS was generated with a popular architecture reverse engineering tool, and metrics were obtained such as lines of code (LOC), McCabe complexity, and program cohesion. A component & connector view is developed which comprises knowledge about the internal structure of the system, and the framework and techniques used. The Allocation view presents the mapping of software on the hardware, in order to provide relevant knowledge for deployment and maintenance. The developed maintainability strategy is used in the next chapter to validate the applicability of the architecture documentation and to identify opportunities for improving the maintainability of the LBCMS.

Chapter 7

Results and Conclusion

The previous chapters describe what knowledge is retrieved from the LBCMS and how this knowledge is captured in the software architecture documentation (appendix A). The goal of developing the documentation was to reduce the maintenance problem described in chapter 3, by eliminating several causes of the problem. In this chapter, the contribution of the documentation to enhance the maintainability of the LBCMS is discussed. First, the correctness and completeness of the documentation is validated. Second, the applicability of the developed documentation for maintenance of the LBCMS is discussed. Finally, a discussion about the results and the conclusions of this research conclude the thesis.

7.1 Validation of the documentation

Correct and complete documentation is imperative for the applicability of the documentation. If the documentation is incorrect, decisions made based on the documentation could be wrong. If the documentation is not complete enough, it is more difficult to fully comprehend the knowledge communicated by the documentation. Validating the documentation is achieved by:

- Testing the feature descriptions of the LBCMS to verify if the descriptions are correct and complete.
- A review workshop with stakeholders to validate the business goals, quality requirements and design decisions for validity and completeness.
- Validating the McCabe complexity [McCabe 76] index with different tools.

Testing the Feature descriptions

In order to determine if the features are documented correctly and completely, a test scenario is produced and executed. The test scenario comprises each feature and a short list of questions which need to be answered while the test is carried out. The execution of testing the feature descriptions is straightforward: each feature in the feature list is tested and the questions of the test scenario are answered. If the questions are answered positively, the feature description is considered correct. Otherwise, the feature is annotated with the concerns that are discovered and the feature description is improved. By retesting the feature the completeness and correctness is validated. In this research, it took three tests before the current list of features was accepted.

The conclusion of testing the features is that the descriptions, sources, and dependencies of the captured features are correct. The documented rationale of a feature comprises a reasonable and realistic justification (and consequent impact) of the feature. However, the possibility remains that other (precedent and present) justifications are overlooked. It would be naive to argue that the

rationale is exclusively correct, but it is correct enough to provide guidance for stakeholders to understand why the feature is implemented in the system.

It is difficult to determine if the list with features is complete. For instance, feature 12 “Insite edit” was discovered by coincidence during the test. Earlier tests and source code browsing proofed insufficient to discover this feature. If a feature is not discovered, it does not mean that the feature does not exist. However, it is not absolutely crucial to obtain an all-inclusive list of all features. For example, technical consultants use the list to compare the features of the LBCMS to those of competing systems. Provided that the list includes the major capabilities of the system, the completeness of the list is less important. The list of features is complete *enough* to understand the major capabilities of the system and the dependencies between the included features and business goals.

Review workshop part 1

The goal of the first part of the workshop was to inform the stakeholders about the documentation and to validate the business goals, stakeholder concerns, vision, and quality requirements. The workshop was organized at Lost Boys. Two technical consultants, two software engineers, and one project manager participated. A script is developed in which the structure of the workshop is presented (appendix B). The following items were validated during the first part of the workshop.

- Business goals; discussion about the context and the primary business drivers.
- Major stakeholders; discussion about the roles of the participating stakeholders with respect to the LBCMS.
- The quality requirements; discussing the architectural drivers and the major quality attributes that shape the architecture.

Validating results in a workshop is a complicated undertaking. In a limited amount of time, stakeholders need to understand the system and its context in order to assess if the documentation is correct and complete. Furthermore, stakeholders have responsibilities regarding other projects, stakeholders are often biased, and often use a different “frame of reference” when talking about, for example, software quality. Based on this observation, it is recommended that other techniques (such as interviews) are also used to validate business goals and quality requirements.

The quality requirements scenarios for performance and availability were quickly validated, because they originate from an existing specification (SLA). Validating other quality requirements proved to be difficult during the workshop; stakeholders are not familiar with quality scenarios (quality requirements are often overlooked in requirements documents at Lost Boys). In order to determine if the requirements are correct and complete, the current architecture of the LBCMS was examined and desired changes to the architecture are discussed. Since an architecture is critical in the realization of qualities, it is possible to determine if the quality requirements are correct by looking at the current architecture and the desired changes to this architecture. Through inspecting the mapping of the features on software components, the support of the quality requirements becomes clear and the quality requirements can be validated.

Although the workshop did not raise uncovered issues, it is unrealistic to conclude that the business goals, stakeholder descriptions and quality requirements are entirely correct and complete. The possibility remains that other stakeholders would point out missing issues, or that certain issues are unknown to everyone and remain unidentified.

Validating the McCabe Complexity

Valuable information about the LBCMS was retrieved by using automated metrics. The goal of testing the obtained numbers is to validate if this information can be treated as factual. Two tools were selected to validate the metrics obtained with Eclipse Metrics; MetricsReloaded¹ and Classycle². The LBCMS source code was analyzed with each tool and the results were captured in a worksheet. The results of the tools are consistent with the information depicted in the documentation. This provides confidence to assume that the information is correct.

7.2 Assessment of the Applicability of the Documentation

The previous section demonstrates that the documentation is correct and complete enough to provide comprehensive and accurate information about the LBCMS. This section discusses the applicability of the documentation for stakeholders during maintenance activities. The applicability of the documentation is validated as follows:

- A review workshop with stakeholders, in which two maintenance scenarios are used to determine if the documentation offers enough understanding to assess where and how the change should be made.
- Developing a strategy for maintainability based on the documentation, in which current maintainability tactics are presented and new tactics are introduced.
- Reasoning about the use of the documentation. In chapter 3, the personas John Foo and Miranda Bar were introduced. In section 7.3, the applicability of the documentation is assessed against the information needs of John and Miranda.

Review workshop part 2; architecture evaluation

“An architecture evaluation should be part of every architecture-oriented development methodology” Bass et al. (1998). Using a structured method such as ATAM or CBAM for the evaluation of the architecture is a relatively cheap and powerful risk mitigation activity. The architecture documentation produced in this project can be used for review activities. This is demonstrated during the workshop, in which stakeholders were asked for scenarios to assess the architecture (adapted from the ATAM). Two scenarios were thought of: 1) what is the impact of replacing the EJB by POJO (Plain Old Java Objects)? And 2) how easily can changes be made in the

¹ A plug-in for the IntelliJ IDE, source: <http://plugins.intellij.net/plugin/?id=93> (june, 2006)

² An open source metrics tool, source: <http://classycle.sourceforge.net/> (june, 2006)

XSLT/Taglib template component? The fact that scenarios could be produced was already illuminating for the stakeholders. Within 15 minutes, stakeholders had determined where and how the changes should occur. It was agreed that the architecture documentation helped to understand the internal structure of the system and that this understanding made it possible to make fast and more accurate decisions.

Maintainability strategy assessment

A new architectural strategy was developed which incorporates maintainability tactics described in literature. The aim of using this strategy is bipartite: 1) the strategy serves as a platform to develop tactics which could reduce the causes of the maintenance problems, and 2) to verify if the consequences of implementing the tactics could be assessed in the views. During the workshop, the strategy was presented and assessed against the LBCMS. The tactics which are not implemented are depicted in red (see Figure 18).

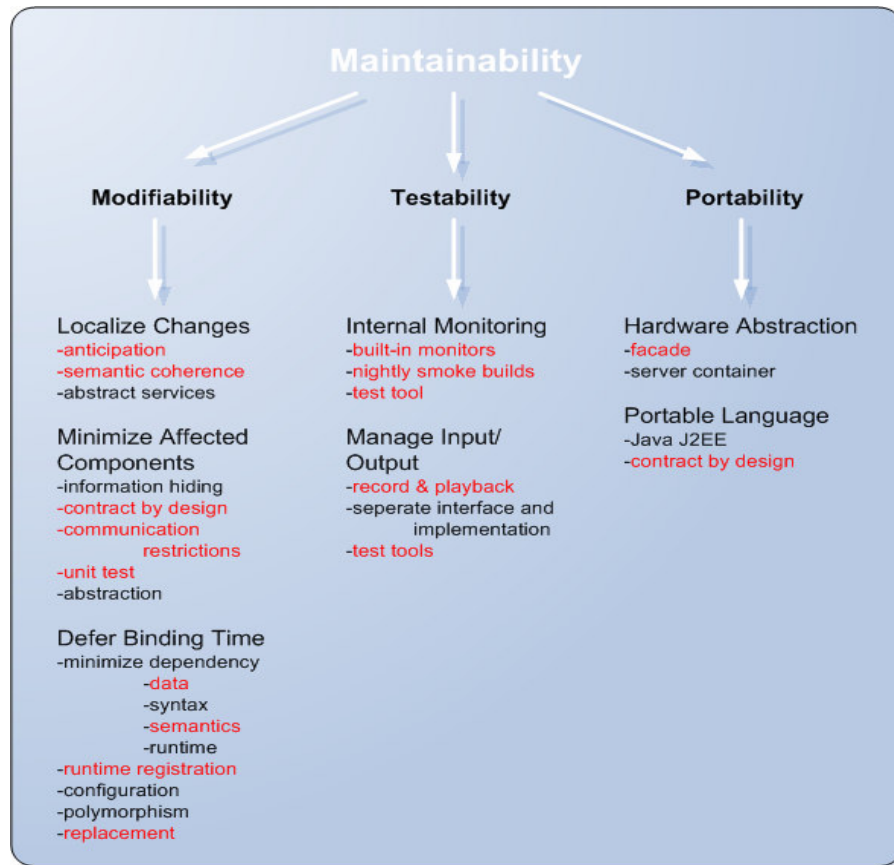


Figure 18 Reasoning about maintainability tactics

An interesting observation was made during the reasoning about the LBCMS based on the documentation and tactics; stakeholders agreed that the design of the system was better than they always believed. Although some parts of the system are too complex and not captured in the

documentation, the documentation helped the stakeholders to develop a mental model of the system which helped them to understand the system. Stakeholders were able to determine where changes should occur if the maintainability strategy presented in Figure 18 is implemented. In addition, stakeholders became aware that the testability of the LBCMS is worse than they had imagined.

7.3 How the documentation supports the tasks of John Foo and Miranda Bar

In Chapter 3, the personas John Foo and Miranda Bar were introduced. John is responsible for maintaining the system and Miranda is responsible for advising customers and making decisions about changes. The problems they encounter with the LBCMS relate to tacit knowledge, poor understandability, and unclear capabilities of the system. In this section, the documentation is validated by discussing how the documentation satisfies the information needs of the personas John Foo and Miranda Bar.

Obtain sourcecode and a working instance

When John needs information to obtain a working instance of the LBCMS, he first consults the Component & Connector view presented in the documentation. The view encompasses information regarding the techniques used within the system. When John desires to deploy the software on hardware, he consults the Allocation view. This view depicts how the software is deployed on the hardware. A test was conducted to determine the amount of time saved by providing this information in documentation. A comparison was done between the time needed to install the LBCMS with documentation and without the documentation. The test pointed out that the time needed to obtain a working LBCMS is reduced from 4 hours to 2 hours. However, more tests are needed to establish a more confident result.

Understanding the goals, internal structure and capabilities of the software

Almost every time when John or Miranda is working with the LBCMS, they need information about the LBCMS in order to understand the system (context, goals, internal structure etc.). The documentation helps John and Miranda to understand the system and its context, by providing comprehensible information about business goals, vision, design decisions, and architectural views.

The documented *business goals* specify unambiguously the goals the customer and Lost Boys have with the system. The business goals are important to John and Miranda, since they constrain the work that is done during maintenance, but also guide stakeholders when new requirements arrive or changes need to be made.

The *architectural views* help John to develop a mental model of the software, without the expensive task of browsing the source code. The views and viewpoints give a clear overview of the system from a certain perspective and provide the information necessary for John to reason about the system with other stakeholders.

The *feature descriptions* in the documentation describe the capabilities of the LBCMS. Miranda uses these descriptions when a *short-list* of competing CMS-like systems is produced.

With the documentation, John and Miranda have explicit and transferable knowledge of the system and no longer depend solely on oral communication and sourcecode browsing in order to understand the system.

Make accurate decisions and change the software

When the design needs to be changed by John, John and Miranda consult the design decisions and the architectural views to make better decisions regarding 1) where the changes should occur, 2) how the change should be made, and 3) what impact the change has (time, resources, correctness and quality). A *vision* is developed and the context of the LBCMS is documented. When Miranda needs to make decisions about, for example, the incorporation of a new feature, she can consult the vision in order to determine whether the feature is desirable with respect to the context and vision of the LBCMS. The *design decisions* guide John and Miranda when changes need to be made; why was the decision made and what impact did the decision have on the architecture of the LBCMS. The documented dependencies between business goals, features and quality requirements help John and Miranda to assess whether the possible solutions are feasible. The documented features help John with the verification if the system integrity has been preserved after the change is made. The feature descriptions in the documentation are manageable through versioning, so future changes in the software can be documented.

7.4 Discussion

The previous sections focus on the validation of the contents of the documentation and the applicability of the documentation for maintenance tasks. However, it is interesting to put the results in a broader perspective with respect to software maintenance. Reducing maintenance costs and enhancing the quality of the LBCMS are two important goals for software companies such as Lost Boys. How can the results of this research contribute to achieve these goals?

Does the documentation reduce maintenance costs?

In order to assess if the documentation can reduce maintenance costs, the costs of maintenance need to be clear. Unfortunately, it was not feasible during this research to obtain a clear picture of all costs involved. Consequently, it was impossible to validate if the documentation contributes in reducing the costs of maintenance. What becomes clear is that the documentation developed during this project helps stakeholders with (a mutual) understanding (of) the goals, capabilities, and internal structure of the system. Decisions can be made with more confidence because the imposed constraints and opportunities of business goals and earlier decisions are documented. Furthermore, the documentation serves as a valuable reference for stakeholders to determine where and how the change should be implemented. In addition, less time is needed for a maintenance engineer to gain an abstract mental model of the LBCMS. The stakeholders of Lost Boys agree that this will eventually reduce costs because less time is needed to make accurate decisions and changes. However, further research is needed to support this theory.

Does the documentation enhance the quality of software?

Does the documentation developed during this project enhance the software quality of the system? That depends on how quality is defined. For example, if the single most important quality of a system is its maintainability, one could argue that documentation enhances the quality of the system. But as described in this thesis, quality is defined by a set of quality attributes. Documentation by itself cannot possibly enhance all quality attributes of a system. It does support quality attributes such as maintainability and modifiability, but the enhancement of other qualities should come from the people who maintain the system. Arguing that this project enhanced the quality of the LBCMS would be unrealistic. However, before the documentation was developed, it was uncertain if a modification had a positive or negative consequence on the quality of the software. When the documentation is seen from this perspective, it does enhance the quality of the decisions and changes made during maintenance by Lost Boys employees. Ultimately, the quality of the software can improve by refactoring existing code based on definite architectural decisions. Future work at Lost Boys is necessary in order to ensure that the actual architecture and its representation in the documentation remain consistent.

Are we done yet?

With the results validated and discussed, one final question remains; “*is the documentation complete?*” This question remains difficult to answer. The possibility remains that important knowledge, such as design decisions, remains tacit. However, by asking “*what is missing?*” several issues come to mind. For example, a procedure at Lost Boys which addresses the maintenance of the documentation is missing. Also, the documentation does not encompass new requirements of customers. Or, the views do not illustrate where frequent changes occur. These are examples of issues which ideally should be addressed in order to fully benefit from the developed documentation. However, it is important to realize that retrieving architectural documentation is a complex and expensive undertaking. Careful considerations are needed in order to determine what knowledge should be retrieved. The answer to the question if the documentation is complete would be *no*. Given the goals of this research, the budget, available resources and other constraints, a realistic answer is that the documentation is *complete enough* to meet the terms of its purpose.

Recommendations for Lost Boys

The motivation for retrieving and documenting architectural knowledge of the LBCMS springs of this need of Lost Boys to improve the maintainability of the system. The documentation provides a platform which makes further improvements possible. The strategy discussed earlier presents several tactics which can enhance the maintainability of the LBCMS. The implementation of a testability strategy would be an excellent starting point. Also, the complexity view illustrates several problems within the current implementation of the LBCMS. For example, revising the large classes in the EJB component could enhance maintainability. The documentation requires maintenance; Lost Boys needs to integrate maintenance of documentation with the maintenance of the LBCMS.

7.5 Conclusion

It goes without saying that writing a document is a difficult task. Writing practical documentation for a software system which is built several years ago is even harder due to the complex nature of software and the evolution of software. What information can be retrieved? Is it valuable? How should the information be presented? When is the documentation satisfactory? These questions are difficult to answer, but need to be asked in order to obtain accurate and usable documentation. In this research, an approach for developing documentation is presented. The aim of the documentation is to provide an abstraction of the architecture and capabilities of the LBCMS for important stakeholders at Lost Boys. In order to obtain the right knowledge and present it in documentation, three research questions were answered in the previous chapters of this thesis;

1. *What maintenance problems need to be solved and how can software architecture documentation solve it?*
2. *When is the documentation of the system satisfactory?*
3. *How is knowledge of the system retrieved in order to capture it in documentation?*

Precise and accurate documentation is needed in order to *understand* the system and to assess consequences of changes. In contrast, documentation is never complete, because it is impossible to cover all concerns and details. However, it still can capture *enough* knowledge which is relevant and useful for maintenance. Based on the problem analysis and methods proposed in literature, the decision was made to reconstruct architectural knowledge and capture it in documentation.

In Chapter 2 and Chapter 3, the landscape of software maintenance is discussed and a problem analysis is presented in order to answer the first research question. The problems which need to be addressed by the documentation relate to tacit knowledge, understanding of the software, and capabilities of the LBCMS. The architecture documentation developed during this research encompasses this knowledge and can be used to:

- Understand goals of the LBCMS
- Understand stakeholder requirements and concerns
- Understand features and requirements and their relationship with the goals
- Determine most important quality requirements
- Assess the impact of changes, support important decisions
- Provide a transferable abstraction of a system
- Enable reasoning about the quality and capabilities of the system, in order to achieve better perfective and adaptive maintenance.

To answer the second research question, a hypothesis was formulated based on the problem analysis and background research:

The Software Architecture Documentation is complete enough when:

- 1) It provides explicit and transferable information about features, quality requirements, design decisions, and architectural views;*
- 2) It visualizes relevant views on the system to support the maintenance engineer' perceptual ability for interpreting complex information about the LBCMS;*
- 3) It enables abstract reasoning about the architecture by stakeholders, in order to identify issues, make rational decisions, and locate where changes should occur.*

The hypothesis was confirmed by analyzing and validating the correctness, completeness and applicability of the documentation. The validation of the documentation included review sessions, a workshop, and reasoning about the applicability by discussing the information needs of two personas. During the review sessions and the workshop, stakeholders agreed that the knowledge the documentation communicates is complete enough to understand the system based on the criteria mentioned in the hypothesis. However, it is still difficult to assess whether the architecture documentation is complete enough, because it remains difficult to decide which information is relevant. In addition, the LBCMS was poorly documented and the code is complex, so the possibility that important information remains unnoticed resides.

The third research question is difficult to answer. How information should be retrieved depends on what information is needed during different maintenance activities. In addition, there are no straightforward methods to retrieve relevant knowledge. The approach presented in Chapter 4 proved to be successful in this project. The approach comprises the retrieval of high-level goals and stakeholder concerns, analyzing the observable behavior in order to capture features, and analyzing the internal structure of the system in order to capture architectural views from the perspective of software maintainability. The relevance of the gathered knowledge is assessed against the maintainability requirements. An architectural strategy (with modifiability, testability and portability tactics) is developed which can be implemented in the system to deal with the (sub) problem. The strategy was used during a workshop to demonstrate possible solutions. Questions about the technical consequences could be answered by examining the views. Stakeholders agreed that the feature descriptions and business goals helped to understand the capabilities of the system. This enabled reasoning about the impact of implementing new features or modifying existing features. In addition, the documented dependencies between goals, features, software, and hardware made the documentation comprehensible and verifiable. The documentation provides a valuable platform to realize future enhancements of the system

References

- [Bass 03] Bass, L., Clements, P., and Kazman, R. 2003. *Software Architecture in Practice, second edition*. Addison-Wesley Longman Publishing Co., Inc.
- [Bennett 00] Bennett, K. H. and Rajlich, V. T. 2000. *Software maintenance and evolution: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ACM Press, New York, NY, 73-87.
- [Boehm 83] Boehm, B. 1983. *Seven Basic Principles of Software Engineering*, The Journal of Systems and Software 3.3-24.
- [Bosch 04] J. Bosch. 2004. *Software architecture: The next step*. EWSA, LNCS 3047, pp. 194-199, 2004. Springer- Verlag Heidelberg.
- [Clements 02] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. 2002. *Documenting Software Architectures: Views and Beyond*. Pearson Education.
- [Dudney 02] Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., and Osborne, D. 2002. *J2EE Antipatterns*. 1. John Wiley & Sons, Inc; ISBN 0471146153
- [Fowler 99] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co; ISBN 0201485672
- [Glass 89] Glass, R. 1989. *Software maintenance documentation*. In Proceedings of the 7th Annual international Conference on Systems Documentation (Pittsburg, Pennsylvania, United States, November 08 - 10, 1989). C. Association for Computing Machinery, Ed. SIGDOC '89. ACM Press, New York, NY, 99-101.
- [Hilliard, 1999] Hilliard, R., 1999, *Using the UML for architectural description*, In Proceedings of UML'99 The Unified Modeling Language, Second International Conference, Lecture Notes in Computer Science volume 1723.
- [IEEE 90] IEEE 1990. *Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, NY, USA, IEEE Computer Society Press.
- [IEEE 93] IEEE. 1993. IEEE Std. 1219: *Standard for Software Maintenance*. Los Alamitos CA., USA. IEEE Computer Society Press.
- [ISO 95] Int. Standards Organisation. 1995. *ISO12207 Information technology - Software life cycle processes*. Geneva, Switzerland.

- [Kang 90] Kang, K., et al. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.
- [Kruchten 95] Kruchten, P., 1995. *The 4+1 View Model of Architecture*. IEEE Softw. 12, 6 (Nov. 1995), 42-50.
- [Kruchten, 1998] Kruchten, P., 1998, *Modeling Component Systems with the Unified Modeling Language*, A position paper presented at the International Workshop on Component-Based Software Engineering.
- [Kruchten, 2004] Kruchten, P., 2004, *An ontology of architectural design decisions in software-intensive systems*, SVM workshop.
- [Lauesen 02] Lauesen, S., *Software Requirements, Styles and Techniques*, Addison Wesley Publishing Company, Harlow (England).
- [Lethbridge 03] Lethbridge, T. C., Singer, J., and Forward, A. 2003. *How Software Engineers Use Documentation: The State of the Practice*. IEEE Softw. 20, 6 (Nov. 2003), 35-39.
- [Lientz 80] Lientz B. P., Swanson E. B. 1980. *Software Maintenance Management*. Addison Wesley, Reading, MA.
- [McCabe 76] McCabe, T. J. 1976. *A complexity measure*. In Proceedings of the 2nd international Conference on Software Engineering. International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 407.
- [McConnell 04] McConnell, S. 2004. *Code Complete, second edition*, Microsoft Press; ISBN 0735619670
- [Muller 00] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M., Tilley, S. R., and Wong, K. 2000. *Reverse engineering: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering. ICSE '00. ACM Press, New York, NY, 47-60.
- [Niessink 00] Niessink, F. and van Vliet, H. 2000. *Software maintenance from a service perspective*. Journal of Software Maintenance 12, 2 (Mar. 2000), 103-120.
- [Nuseibeh 00] Nuseibeh, B. and Easterbrook, S. 2000. *Requirements engineering: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM Press, New York, NY, 35-46.

- [Parnas 01] Parnas, D. 2001. *Software aging*. In *Software Fundamentals: Collected Papers By David L. Parnas*. Addison-Wesley Longman Publishing Co., Boston, MA, 551-567.
- [Rajlich 00] Rajlich, V. T. and Bennett, K. H. 2000. *A Staged Model for the Software Life Cycle*. *Computer* 33, 7 (Jul. 2000), 66-71.
- [Rich 90] Rich, C. and Wills, L. M. 1990. *Recognizing a Program's Design: A Graph-Parsing Approach*. *IEEE Softw.* 7, 1 (Jan. 1990), 82-89.
- [Tilley 00] Tilley, S. R. 2000. *The canonical activities of reverse engineering*. *Ann. Softw. Eng.* 9, 1-4 (Jan. 2000), 249-271.
- [Walz 93] Walz, D. B., Elam, J. J., and Curtis, B. 1993. *Inside a software design team: knowledge acquisition, sharing, and integration*. *Commun. ACM* 36, 10 (Oct. 1993), 63-77.
- [Wieggers 04] Wieggers, K.E. 2003. *Software Requirements, 2nd Edition*, Microsoft Press; ISBN 0735618798

Appendix A
Software Requirements and Architecture Documentation

Appendix B
Workshop presentation