A Systematic Way to Develop the Software Architecture based on Architecture Vision
This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Muhammad Aamir Faried
Mustafa Ilyas
E-mail: aamirfaried@gmail.com
E-mail: Mustafa_ilyaspk@hotmail.com

University advisor(s):
Dr. Mikael Svahnberg
Assistant Professor
Department of Systems and Software Engineering
Blekinge Institute of Technology
School of Engineering
PO Box 520, SE - 372 25 Ronneby
Sweden

School of Engineering
Blekinge Institute of Technology
Box 520
SE - 372 25 Ronneby
Sweden
Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25
In the software development life cycle, changes are inevitable. Designing the architecture of the software and writing the source code does not end the software life cycle. The software system evolves as changes in the environment and requirements are incorporated in the system. If these changes are not managed properly, the architecture of the software deteriorates and leads to architecture erosion. This study is an effort to address the problem of architecture erosion and to keep the software architecture live and up to date. In this study we propose a method to minimize or avoid architecture erosion. This method provides a systematic way to design the architecture of the software system. It is based on the concept of architecture vision that is the ideal representation of the architecture of the software in hand. This method treats architecture document and architecture design document as two completely different documents. Moreover this study presents an effort to establish a relationship between the implemented architecture of the system, the architecture design document and an architecture vision. The dynamic validation of the devised method shows that this method is suitable for medium to large scale projects that have several releases. All the activities of this method revolve around the efforts to keep the architecture of the software aligned with an architecture vision. The iterative nature of the method and synchronization of the implemented architecture with the architecture vision helps to detect and reduce architecture erosion. This method does not conflict with or replaces any development or management process or method (like RUP, agile, water fall etc), but exists parallel to them.

**Keywords:** software architecture, architecture erosion, architecture drift, architecture vision.
ACKNOWLEDGEMENTS

We are thankful to Almighty Allah for giving us the courage and ability to finish this thesis. Without His blessings we would not have been able to finish this research work.

We are extremely thankful to our advisor Dr. Mikael Svahnberg who has been a motivating factor for us and his expert opinion throughout our research were guiding force for us. He has been extremely patient during all this time and provided invaluable feedback and support during every step of this thesis work.

Lastly we would thank our families who were with us during our studies and were the key source of motivation.
# Table of contents

Chapter 1: Introduction .................................................................................... 6
  1.1 Research Gap ................................................................................ 8
  1.2 Aims ............................................................................................ 8
  1.3 Objectives ..................................................................................... 8
  1.4 Research Questions ........................................................................ 8
  1.5 Expected Outcomes ........................................................................ 8
  1.6 Research Methodology .................................................................... 9
  1.6.1 Literature review ........................................................................... 9
  1.6.2 Qualitative analysis: ....................................................................... 9
  1.6.3 Dynamic validation: ..................................................................... 10
  1.7 Mapping of Research Questions to Research Methodology .......... 10
  1.8 Architecture Erosion ..................................................................... 19
  1.9 Architecture Documentation .......................................................... 19
  2.1 Goals of V-BAM ............................................................................ 20
  2.2 Pre-requisites .............................................................................. 20
  2.3 Input / Output of the V-BAM .......................................................... 20
  2.4 V-BAM Activities ........................................................................... 21
  2.5 Summary .................................................................................... 28

Chapter 3: V-BAM Validation ........................................................................... 30
  3.1 Company .................................................................................... 30
  3.2 Product ....................................................................................... 30
  3.3 Method of Getting Data ................................................................. 30
  3.4 Problems in the CIS Product .......................................................... 30
  3.5 Application of V-BAM .................................................................... 32
  3.6 Experiences from Applying the V-BAM Method ......................... 35
  3.6.1 Advantages ................................................................................. 35
  3.6.2 Disadvantages / Restriction ........................................................... 37
  3.7 Scalability of V-BAM ..................................................................... 37
  3.8 Validity Threats ........................................................................... 38

Chapter 4: Discussion ..................................................................................... 40

Chapter 5: Conclusion .................................................................................... 42
  Conclusion: ................................................................................................ 42
  Future Work ............................................................................................... 42

References .................................................................................................... 43

APPENDIX A .................................................................................................. 46

APPENDIX B .................................................................................................. 48

GLOSSARY .................................................................................................... 61
Chapter 1: Introduction

In the software development life cycle changes are inevitable, it is a process of continuous modeling and refinement. Designing the architecture of a software system and writing the source code does not end the software life cycle. The software system evolves with the changes in the environment, requirements and hardware [1, 2, 3, 12]. Lientz and Swanson [29] categorized the maintenance and evolution activities into four classes that are adaptive, perfective, corrective and preventive. Adaptive refers to the changes in the software environment whereas perfective covers new user requirements; corrective is about fixing errors and preventive is the prevention of problems in the future [29]. The maintenance and evolution activities are most expensive activities, they consume about 60% to 80% of the total efforts spent on a software product [10]. The overall objective of evolution it to modify the existing software product while preserving its integrity. Ned Chapin et al [30] define evolution as the application of software maintenance activities and processes that generate a new operational software version, together with the associated quality assurance activities and processes, and with the management of the activities and processes. The software systems are critical business assets and to maintain the value of these assets they must be changed, updated and managed. In business terms, the software evolves because it is successful in the marketplace, user demand is strong, the development atmosphere is vibrant and positive and return on investment is excellent [15].

As the software evolves, its size and complexity increases. Optimally, the architecture must scale up to support changes in the requirements. If these changes are not managed properly, the architecture of the software deteriorates [2, 4, 6, 7, 8]. Parnas discuss the phenomenon of software deterioration and describe it as software aging [4]. Parnas identifies two major reasons of software aging: failure of the software system to meet changing needs and second is the result of the changes that are introduced to the system [4]. Sutirtha Bhattacharya and Dewayne E. Perry in [7] suggest that if software evolution becomes inevitable, then it is important that an assessment of system evolution should be developed at the architecture level. In [14] N.Subramanian and L.Chung concluded that evolution at the architectural level can be most critical to the success and survival of the pertaining software system, as it is at highest level of the solution.

In the software evolution process, the software architecture takes key role. The architecture is meant to guide the evolution process, while also being evolved [19]. It has been reported that as the software evolves its architecture decays [20]. Architecture erosion is defined as a phenomenon in which an applications initial architecture is modified arbitrarily to the point where it no longer holds its key properties [31]. According to Jaktman et al in [27] erosion is caused by an increase in the complexity of architecture, an unclear relation between the architecture document and its implementation, an increase in the defect rate and unpredictable behaviour due to modifications. There are a several problems that cause erosion. Most of these problems are associated with the way software is commonly developed and maintained. Van Gurp and Bosch [1] identifies some of these problems like; most iterative development methods allow to incorporate new requirements that have an effect on the architecture, complex notations are used to develop software that makes it difficult to track design decisions and these design decisions are normally conflicting and overlapping. Moreover, developers also take suboptimal design decisions during development and maintenance [12]. Apart from the
problems in development methods, there are number of other problems that cause erosion such as violation of architecture design description during implementation, changes in the design description without considering its impact [2], deviation from the original idea of the architecture design, inconsistency between the architecture design description and its actual implementation [4].

The architecture of the software moves away from its original shape if the architecture is changed without taking into account the impact of the change. This will naturally result in a drift of the architecture, as there is no longer an updated architecture to relate changes to [4]. This situation eventually leads to architecture erosion. Software architecture erosion and drift both affect the ability of the architecture to fulfill quality and functional requirements of the system [4]. Over a series (n) of releases, the architecture of the next release (n+1) is designed based on the previously eroded architecture that raises questions on the validity of the designed architecture. After a couple of iterations, this leads to a situation where the architecture document exhibits overlapped, vaporized, obsolete and accumulated design decisions [4]. Architecture erosion and drift can be ascribed to the same causes [4]. In this research [4], a distinction is made between architecture erosion and drift. Architecture erosion means violations to the architecture, whereas architectural drift means a loss of the original idea when changes are incorporated in the system [4, 2].

As a result of architecture deterioration the release pace is affected and features arrive late in the market [34]. Architecture erosion can lead to a situation where the architecture may have to be prematurely retired. This will delay the release of the product even more, which further undermines the value of the product. The best way to overcome the negative effect of software aging is to place software change and evolution at the centre of the software development process [21, 22]. Ideally development of software systems starts with requirements through architecture to the implementation. Any subsequent changes to the software system should follow the same path. However, in most of the cases neither the software development nor the evolution and maintenance process follows such a path. Due to several business or technical constraints these changes are mostly introduced directly into the implementation.

A common technique to address system erosion and drift is to keep the software architecture document up-to-date [11, 12]. Van Gurp & Bosch in [1] identified that only few software development organizations actually manage to keep the architecture document up-to-date. In most of the cases, the architecture document is developed at the start of the project and is not maintained properly during the development life cycle. It leads to a situation where source code is left as the only available up-to-date documentation. Based on this, it can be inferred that the architecture of the software is not the architecture that is designed in the architecture document; it is the architecture that is actually implemented. There are two types of architecture documents, one that is used as a means for communication whereas the other is used to define overall goals to aspire for in a particular release. Often, the architecture document contains a mixture of overall goals of the system and detailed design. The solution to this problem is to keep two thoroughly different documents. One document contains the overall architecture of the system that can serve as an architecture vision while other document contains more detailed architectural design decision for a particular release.
The Architecture Vision is the document that represents overall goals of the software system at a high level of granularity, whereas the architecture design document represents an instance of the vision for a particular development release. The architecture vision represents an ideal architecture of the system that satisfies functional and quality requirements in an ideal way. This ideal picture serves as a goal to inspire and provide guidelines and motivation for the architecture in next release(s). Whether it is actually possible to achieve these goals is not that important; at least we have some ideal goals to strive for. An architecture vision helps to keep the architecture document up-to-date and to increase the ability of the architecture to fulfil its quality goals for the next release.

1.1 Research Gap
Now we have reached a point where on one end we have currently implemented software architecture and design document to define the architecture of current release and on the other end we have architecture vision that explain how the current and future architecture ideally should look like. Still not much work has been performed to identify relationships between the current architecture of the system, the architecture design document and an architecture vision. In addition what is required is to have a method that will minimize the efforts to design the architecture for the next release in the light of the architecture vision, the current architecture of the software and the architecture document for the current release. As the software evolves, we need to keep the architecture vision consistent with the requirements, and thus we need to update the architecture vision.

1.2 Aims
The main objective of our study is to devise a method to reduce or limit architecture erosion. This method will help to minimize the efforts to design the architecture for (n+1 release) on the basis of the current architecture (under ‘n’ release), the architecture design document and an architecture vision.

1.3 Objectives
The following objectives are set to meet the goal:

1. Identify how erosion can affect the architecture of evolving software.
2. Identify current state of research to overcome architecture erosion.
3. Determine the importance of an architecture vision and identify how it can be helpful to reduce or limit architecture erosion.
4. Devise a method to minimize the effort required for designing software architecture for the next iteration.

1.4 Research Questions
How can we minimize the efforts to design the architecture for (n+1 release) on the basis of the current architecture (under ‘n’ release), the architecture design document and an architecture vision?

1. How can erosion affect the architecture of evolving software?
2. What is the current state of research to overcome architecture erosion?
3. What is the architecture vision, how would it help to reduce or limit architecture erosion?
4. How the architecture vision can be used to reduce the architecture erosion?

1.5 Expected Outcomes
The outcome of this thesis is a report that will discuss following points:
1. A method that will help to reduce or limit architecture erosion and based on the current architecture, the architecture design document and an architecture vision.

1.6 Research Methodology
A combination of qualitative and quantitative research methodologies will be used.

1.6.1 Literature review
Literature Review is a method of identifying, evaluating and interpreting all available research relevant to a particular area. Three main reasons to perform a detailed and comprehensive literature survey are:

- Literature survey will provide the basis for understating the architecture erosion and causes of the architecture erosion.
- The literature survey will help us to identify the current state of research to limit and reduce the architecture erosion.
- It will help to identify the relationships between the current architecture, the architecture design decision and an architecture vision.

We will only use the authentic databases to perform the literature review like IEEE Xplore, Springer Link, and ACM Digital Library. The search string that we will use in the research is mentioned below. The main purpose of the literature review is to identify what exactly is architecture erosion, its causes and effects, and what has been done to overcome the architecture erosion. Hence the published material that satisfy the said purpose was mainly considered for this study. The data extraction form that is used in this study is presented in the table 1.1.

("Software") AND ("architecture erosion" OR "erosion") AND ("vision" OR "architecture vision" OR "importance" OR "limit" OR "reduce" OR "finish" OR "relationship" "current" OR "current architecture" OR "design" OR "architecture design" OR "architecture decision" OR "architecture design decision") AND (method OR model OR tool OR solution OR technique OR recommendation OR framework OR process OR approach OR practice OR methodology)

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Title of the published paper</td>
</tr>
<tr>
<td>Author</td>
<td>All the authors those are creditable in the area of architecture and architecture erosion.</td>
</tr>
<tr>
<td>Year</td>
<td>Valid Inputs: 2000 – 2009</td>
</tr>
<tr>
<td>Main Empirical Method</td>
<td>Valid Inputs: Survey, Case Study or Other</td>
</tr>
<tr>
<td>Focus of Study</td>
<td>Valid Inputs: Collaboration in General, Single Practice, Development Phase or Other</td>
</tr>
<tr>
<td>Identified causes</td>
<td>Collect the causes of the architecture erosion.</td>
</tr>
<tr>
<td>Purposed Solutions</td>
<td>The solution purposed to overcome the architecture erosion.</td>
</tr>
<tr>
<td>Additional findings</td>
<td>Link between the current architecture, architecture design decisions and architecture vision.</td>
</tr>
</tbody>
</table>

Table 1.1: Data extraction form

1.6.2 Qualitative analysis:
The literature survey will provide the basis for the understanding the causes of architecture erosion, what has been done to overcome the architecture erosion. Based the findings of the literature survey, a theoretical method will be devised that will provide a way to design the architecture for the next release in a way that minimize the architecture erosion in n releases. The devised method is presented in the chapter two of the thesis.

1.6.3 Dynamic validation:
To evaluate our devised method we will perform dynamic validation. In order to perform dynamic validation we will observe how the architecture of an appropriately sized project changes during a couple of releases and test our methodology by maintaining an architecture vision and an updated architecture document for the project. The details and outcome of the dynamic validation is presented in the chapter three of this thesis.

1.7 Mapping of Research Questions to Research Methodology
The mapping of each research question into research steps and research methods is show in the table 1.2.

<table>
<thead>
<tr>
<th>Research Question</th>
<th>Research Step</th>
<th>Research Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ 1</td>
<td>RS1.1: What is architecture erosion?</td>
<td>literature review</td>
</tr>
<tr>
<td></td>
<td>RS 1.2: What are causes of architecture erosion?</td>
<td></td>
</tr>
<tr>
<td>RQ 2</td>
<td>RS 2.1: What are the solutions purposed to overcome architecture erosion.</td>
<td>Literature review</td>
</tr>
<tr>
<td>RQ 3</td>
<td>RS 3.1: What is architecture erosion: RS 3.2: How it will help to overcome architecture erosion</td>
<td>Literature review</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Qualitative analysis</td>
</tr>
<tr>
<td>RQ 4</td>
<td>RQ 4.1: Device a method that will reduce the architecture erosion based on the findings of RQ1, RQ1 and RQ3</td>
<td>Qualitative analysis</td>
</tr>
</tbody>
</table>

Table 2.2: Research steps to answer research question

1.8 Architecture Erosion
In the software engineering paradigm Architecture Erosion is a common problem. Software architectures tend to erode as changes are incorporated in the software. As a result of these changes the software architecture reach the point where redesigning the architecture from scratch is a better alternative than to prolong the existing architecture. A number of studies (e.g., [2, 3, 4, 5]) have been conducted to address the problem of software architecture erosion. These studies propose approaches like architecture patterns, architecture presentation languages, development methods etc.

Conducting literature review:
In order to conduct the literature review below mentioned steps were performed:

Three databases (ACM Digital Library, IEEE Xplore, Springer Link) were searched using the search string mentioned in the section 1.6.1. Total 75 number of papers where found as a result of the search.
We have performed some initial triage, using the data extraction form designed earlier. We have removed total number of 15 papers that were published prior to 2000. An exception is made for Parnas, Perry and Wolf papers. They come up with the quite important causes and effects of the erosion and gives overview of the problem in detail.

Among the total number of 60 papers that was left, there were some duplicates papers as well. Total 5 duplicate papers were removed from the list of papers. Once we have removed the duplicated paper, we have gone through the abstract of the papers to remove any papers that are not really related to the architecture erosion problem and causes. As a part of this step total 15 papers were removed. Using the data extraction form mentioned in the table 1.1 under section 1.6.6. We have read the papers and found that 20 papers do not match the criteria defined in the data extraction form. Ultimately 25 found to be relevant. To make sure that have not overlooked any important point, both author went through the paper separately. Once we have gone through all papers we compared the results and had discussion about the findings. We have than complied the findings of the literature survey and performed the analysis on the basis of that. The output of this step is presented below.

The section below presents some existing studies/efforts that have been conducted to analyze and overcome the causes of architecture erosion. We also discuss the solutions proposed by these studies to overcome architecture erosion.

According to Perry and Wolf [2] evaluation and customization are two factors that are important for software architectures. The accompanying property of evolution is increase in the resistance to change; this resistance is due to two architectural problems: architecture erosion and architecture drift. They argue that Architectural Erosion is a result of violations of the software architecture. The software architecture gets violated because of many reasons like architecture is not clear to the developers, time constraints, no or little documentation etc. These violations lead to an increase in problems in the system and contribute to the resistance to change. They describe Architectural drift as a result of insensitivity about the software architecture. This insensitivity leads to the inadaptability of the architecture to accommodate the changes as a result of this the architecture looses coherence and clarity that interns make it more coupled and complex. This increases the probability to violate the architecture as the architecture lack clarity and becomes more complex.

They have proposed a model for software architectures that consist of three components that are elements, form and rationale. This model emphasizes the architectural elements of data, processing, and connection. Moreover it also highlights their relationships and properties.

Parnas [4] discuss a similar kind of situation or behavior (though he did not talk about erosion explicitly) and termed it as software aging. Parnas argues that software aging is inevitable but we can slow down this process or some time even reverses its effects. Parnas believe that causes of software aging are:

- Changes in the environment (domain, requirement and technology) around the software and these changes are not incorporated in the software.
- Changes are introduced to the system in a careless manner without taking care of their effect (change impact) which degrades the system.
- Changes are not documented properly that lead to the situation where it is hard to understand and make changes in the system.
In this study Parnas describe three major downsides of software aging:

- **Inability to keep up**: It is difficult to incorporate changes or extends the exiting software. In order to make a change you not only have to understand what changes you have to make but also where and how to make these changes.
- **Reduced Performance**: As software grows its architecture deteriorates and the quality requirements like time/space, performance is compromised.
- **Decreasing reliability**: As a result of changes in the software, errors are introduced. New bugs bubbled up as impact of change is not analyzed properly.

To address the problems of software aging Parnas suggests some guidelines or techniques to overcome or limit its effects:

- **Design for Success**: Software applications should be designed by having change factor in mind. It is quite difficult to predict actual changes but we can predict classes of change i.e. Replacing of the terminal with a new type, or changes in the user-interface. We can then organize the software so that items that are most likely to change are placed separately. The principle to be applied is known by different names like 'information hiding', 'abstraction', 'data hiding', 'separation of concerns', 'object orientation' etc.
- **Documentation – keeping record**: Along with code documentation, Design principles and design decision should be documented or recorded in the form that can be useful for future. After having necessary documentation, it is required to keep them up-to-date so that they remain consistent with system.
- **Reviews – more than one opinions**: Every design and other software artifacts should be reviewed and approved by someone whose responsibility are for the long term future of the product.

As the size and complexity of software increase the weaknesses of existing software methods begin to show. **Van Gurp and Bosch** [1] has identified that design erosion is caused by a number of problems associated with the way software is commonly developed.

- **Traceability of design decisions**: Design decisions are difficult to track because notation used lack expressiveness.
- **Increasing maintenance cost**: During evolution and maintenance process developers take suboptimal design decisions. There could be many reasons for these decisions either because developers do not understand the architecture or because a more optimal decision would be too effort demanding.
- **Accumulation of design decisions**: Often design decisions accumulate and interact in such a way that if you revise one design decision you have to revise other design decisions as well.
- **Iterative methods**: Software is designed to accommodate expected changes but iterative methods make it possible to incorporate new requirements that have impact on architecture.
- **Change requests**: This conflicts with the iterative nature of many development methods (extreme programming, rapid prototyping, etc.) since these methodologies typically incorporate new requirements that may have an architectural impact, during development whereas a proper design requires knowledge about these requirements in advance.

**Van Gurp and Bosch** [1] propose two stereotype strategies to incorporate changes in the software in the iterative development methods.
• Minimal effort strategy. This strategy encourages to preserve the old system as much as possible and adjusts the change in next development iterations. The advantage of this strategy is that each subsequent iteration results in relatively less cost.

• Optimal design strategy. Update software artifacts by introducing changes required to develop an optimal system for the new set of requirements. The advantage of this approach is that the changed system is optimal for the requirements because any conflicting design decisions in the previous version are resolved. This means that future changes can be incorporated at a relatively low cost.

S. Trujillo, M. Azanza, O. Díaz and R. Capilla [5] studies the importance of the documentation of software architecture design decisions. They explore how the architecture can be extended on the basis of design decisions in product line architecture. They discovered that the existing approaches to software architecture documentation typically focus on the description of components and connectors. These approaches have no or little focus on the documentation of the design decisions that are made during the architecture development phase. As a result of that Architecture Erode and results in the high maintenance cost of the software.

The authors of [5] believe that the design decisions are important elements and they must be documented explicitly during the architecture development process. The main goal of documenting and representing the architecture design decisions is to bridge the gap between the software requirements and software architecture. They emphasize the reuse of the existing design decisions for the architecture customization and extension.

Medvidovic et al. [6] looks at the problem of the software erosion in the context of software evolution and argues that the software architecture guides the system evolution process. In addition, software architecture is a reification of the system requirements. During the system implementation the software developers frequently deviate from the architecture. As a result of this deviation the real (implemented) architecture is arbitrarily modified to the point where its main properties become difficult to sustain and architecture erosion occurs.

Often software architectural artifacts are not in sync with the system's requirements and its implementation, which results in architecture erosion. They have identified different reasons for this behavior.

• Under short deadlines requirements are implemented immediately, and their change impact and effect on the architecture or low level design is not documented.

• Violation of the architectural design decisions to achieve some non-functional quality requirements like increased performance, satisfy real time constraints, reduce memory usage etc.

• Off-the-self functionality that is directly incorporated into the existing system.

• Existence of legacy code that is perceived to prevent careful system architecting.

The have further identified potential problems associated with the architecture erosion.

• It becomes difficult to assess how well the current implementation satisfies the current system requirements.

• Inability to trace a specific requirement to implementation artifacts.
• Lack of understanding the impact or complex effects of changing a requirement.
• System maintainability and evolve-ability becomes very difficult.
• The incorrect perception of the software architecture may lead to the incorrect architectural and implementation design decisions.

To solve or minimize the architecture erosion, the authors have proposed a technique that combines architecture discovery (from system requirements) and architecture recovery (from system implementation). They assume that the given system requirements and the implementation is available and the architecture relevant information either does not exist or is incomplete. The main goal of their research in [6] is to develop general application and style centric approach in order to integrate and reconciliation the identified differences. Their approach combines three separate but complementary techniques: 1) technique based on the architectural style for the architectural discovery from the software requirements 2) architectural style-based technique for the architectural recovery based on the software implementations, and 3) a technique that leverages styles to merge the results of the discovery and recovery.

L. Ding, N. Medvidovic in [10] looked at the problem of software erosion in context of the Object Oriented Paradigm. They believe that software erosion occurs because of the following factors.
• Object oriented applications are increasingly complex and user driven.
• Applications are developed more rapidly and evolve more frequently.
• Frequency with which the requirements are changed in an application.
• Sloppiness with which the changes are often documented.

They figure out the effects of the architecture erosion.
• Architecture erosion causes major departure from intent of the original architecture and the conceptual integrity.
• It causes discrepancy between the architecture “as documented” and “as is”.

The authors believe that the architectural evolution drives the software evolution process. They proposed an approach called Focus, where the main goal is to enable the effective evolution of applications with minimum efforts by recovering its architecture and using it as basis for the architecture evolution. Using this approach, the architecture is recovered incrementally, i.e. only modify those parts of the application that are affected by the given change and document their relevant architectural characteristics.

Bosch [11] discusses erosion in the context of software product families. He believes that in product families the erosion occurs because of the following reasons.
• Mismatches between the optimal approach and the approach currently used by an organization may leads to the several problems like a high degree of erosion.
• During the product life cycle, in most cases design decisions are taken implicitly instead of explicitly on the basis of some clearly stated objectives.
• Initial development or evolution of the shared components is usually performed without the sufficient consideration of all aspects of the design.
The reason for this is schedule and other pressures on the shared components.

- Shared component contains the product specific functionalities that play the key role in the erosion of the components.

Bosch presents a framework for deciding an optimal model for organizing shared components development. In particular, he introduces some dimensions for different approaches and discusses their relevant advantages and disadvantages. Furthermore, based on a three-stage adoption model, he presents a framework for selecting the best alternative for each decision.

While building a bridge between the architecture and the programming languages, Hubert, et al. [8] argued that ADLs (Architecture description Languages) in the classical software development model are mainly engaged in the analysis and design steps while implementation is only supported by code generation facilities. The authors find the problem that common implementation environments are not able to represent the architectural concepts completely or properly. So during the implementation and maintenance phase, hand-coded implementation tends to lose connection to the architectural ADL model. This condition results in the architecture erosion.

While discussing the solution of this problem they introduce the concept of 'Architecture Programming language' (APL). In the architecture programming language, they propose the inclusion of architectural concepts (like components, ports with provided and required interfaces as well as protocols, connectors, and assemblies) into a programming language.

To prove their concept, they present a new programming language Java/A [9] as an instance for a Java-based architectural programming language. In [8] they show how Java/A integrates architectural notions into Java, and present the abstract component model which forms the semantic basis underlying Java/A. [8]

Bhattacharya and Perry [7] propose a model for tracking software evolution and propose measures that will objectively indicate the extent of deviation or divergence of the architecture from the baseline architecture of the software system. They have categories the different aspects of the software. These are the aspects that are sensitive to change and changes in these aspects can significantly affect the usability and conceptual coherence. They argue that it is common practice that any software without an active roadmap soon falls out of favor within its user community. Further they identify reasons for this slip:

- System requirements evolve
- The context in which the software operate changes
- New capabilities are identified that build on existing ones
- Maintenance activities are performed to keep the software operational.

The authors insist that ideally changes to the system should be rectified in the architecture first and followed by changes or enhancements in the implementation. But because of time to market pressure and sub-optimal development processes, changes to the system often erode the fundamental characteristics of the original architecture. It results in a system that fails to fulfill the quality requirements of the system like reliability, availability, performance etc.
D. Falessi, R. Capilla and G. Cantone in [36] argued that architecture has high cost of change and it may erode during the evolution. In order to prevent the design erosion and knowledge vaporization the design decision and rationale for the design decision should be document explicitly. In this study author have proposed a value based approach for Design Decisions Rationale Documentation (DDRD) that focus on documenting the set of required information based on the purpose of the document. In this study they have divided the information into three categories useful information”, “required information” and “optional information”. The key idea is that information included in the DDRD is dependent on the DDRD use case and different use cases require different category of DDRD.

J. V. Gurp, J. Bosch, S. Brinkkemper in [35] identify that as the system grow it become difficult to make new changes, eventually it reaches the point where it is more feasible to replace it or at least re-factor it. The focus of this study was to identify the causes and effects of design erosion on system and how to overcome and prevent the architecture erosion. They have performed the case study on two projects in one organization, on the basis of the case study they have concluded that design erosion is in evitable.

The system of erosion as identified in this study are Low quality code, Uncertainty about specifications, fixes for the code introduce more problems and Deployment problems. Both projects in this case study share some common causes of the erosion that are: designed decision where not document, during the evolution little attention was given to the design, quick fixes and time pressure. Further in the study they have presented the ways to solve the erosion (Redevelopment, Restructuring, Strong focus on design, take product release cycles into account) as well as techniques to prevent the erosion like automatic regression testing, changes should be document properly and Stronger focus on process.

<table>
<thead>
<tr>
<th>Study</th>
<th>Causes of Erosion</th>
<th>Effect of Erosion</th>
<th>Proposed Solution</th>
</tr>
</thead>
</table>
| Perry and Wolf in [2]  | • Violation of Architecture.                                                     | Inadaptability of architecture as it lost focus and clarity and get more couple and complex/ | • Motivation of architecture specification  
• Software architecture model that emphasize on architectural elements of data, processing and connections and highlights their relationships and properties. |
| Parnas in [4]          | • Changes in the environment (domain, requirement and technology) are not incorporated.  
• Side effect of the changes.  
• Changes are not | • Difficult to incorporate changes or extends the exiting software.  
• Reduced Performance.  
• Decreasing | • Design system with change factor in mind.  
• Design principles and design decision should be documented.  
• Design should be reviewed by |
<table>
<thead>
<tr>
<th>Authors</th>
<th>Problems</th>
</tr>
</thead>
</table>
- Design notation lack expressiveness.  
- Complexity of the system grows.  
- Design decisions are not maintained properly.  
- Iterative methods.  
- Design of the software gets eroded.  
- Make necessary changes to the software artifacts to get an optimal system for the new set of requirements.  
- Incorporate the change in the next iteration of the development while preserving old system. |
| Trujillo .S, et al |  
- Existing software architecture documentation approaches only focus on component and connection  
- Architecture document get eroded and not that much useful.  
- Design decision should be documented explicitly.  
- Reuse of design decision. |
- Software architectural artifacts are not synchronized with the requirements of system and its implementation.  
- Difficult to evaluate current implementation  
- Traceability of the requirement.  
- Change impact.  
- Maintainability. |
| Lei and Nenad |  
- Complexity of application.  
- Rapid development.  
- Frequency of requirement change  
- Sloppiness in documenting change  
- Architecture move away from intent of original architecture. Differences between "as documented" and "as is" architecture.  
- Recover architecture and use it as basis for architecture evolution.  
- Document relevant architecture characteristics. |
| J. Bosch |  
- Mismatch between the approach in use and optimal approach.  
- Implicit design decisions.  
- Impact analysis of shared components is not performed in detail.  
- Architecture erosion.  
- Framework for deciding optimal model for organizing shared components.  
- Framework to select best alternative for each decision. |
<table>
<thead>
<tr>
<th>Authors</th>
<th>Causes of Architecture Erosion</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hubert, et al.</td>
<td>• Short coming in ADL to represent architectural concepts.</td>
<td>• Architecture erosion.</td>
</tr>
<tr>
<td></td>
<td>• Architecture erosion.</td>
<td>• New architectural programming language.</td>
</tr>
<tr>
<td>Bhattacharya and Perry</td>
<td>• System requirements evolve.</td>
<td>• Architecture erosion.</td>
</tr>
<tr>
<td></td>
<td>• Environment of software changes.</td>
<td>• Model to access architecture erosion.</td>
</tr>
<tr>
<td></td>
<td>• Software maintenance.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Changes in the system are not documented and reified in the architecture.</td>
<td></td>
</tr>
<tr>
<td>D. Falessi, R. Capilla and G. Cantone</td>
<td>• System evolution</td>
<td>• Value based design decision rationale Documentation.</td>
</tr>
<tr>
<td></td>
<td>• Architecture erosion</td>
<td></td>
</tr>
<tr>
<td>J. V. Gurp, J. Bosch, S. Brinkkemper</td>
<td>• Vaporized design decisions.</td>
<td>• Automatic regression testing.</td>
</tr>
<tr>
<td></td>
<td>• Too little attention to design during evolution.</td>
<td>• Document changes.</td>
</tr>
<tr>
<td></td>
<td>• Quick fixes.</td>
<td>• Stronger focus on process.</td>
</tr>
<tr>
<td></td>
<td>• Experience.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Time pressure.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Evaluation.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• New requirements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Change of staff.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Low quality code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Uncertainty about specifications.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Regressions.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Deployment problems.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defect rates &amp; cost.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Architecture erosion</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Summary of existing studies on Architecture Erosion

Table 1.3 summarized the existing efforts done to identify the causes of the architecture erosion, problem caused by the architecture erosion and different solutions to overcome architecture erosion. Form the analysis of this literature survey we can deduce that more or less all authors believe that changes in the existing system and sloppiness in documenting the change to the system is among the major reasons of the architecture erosion. If changes to the system and design decisions are not documented properly, it will result in the situation where architecture lacks clarity and coherence. As a result the system gets complex and hard to understand, eventually it leads to the architecture erosion. To solve this problem of architecture erosion most of the studies suggested that as the system evolves, document all changes in the relevant artifacts. Moreover some studies explicitly focus on documenting the design decisions. On the basis of this literature survey, in this study we have purposed a systematic way to develop the software architecture that focus on the documentation of changes in the system. To make the documentation simple we have split the architecture document into two separate documents. One is the vision that contains the system level architecture goals,
objective, design decision etc. and other document contains the release specific software architecture.

1.9 Architecture Documentation

In the software engineering paradigm, documentation is a supporting tool of every artifact. A number of studies [15, 29, 17, 18, 16] has been conducted to address the importance of the documentation during the software life cycle. In the software industry, few software development organizations manage to keep the updated version of the software architecture documentation [32]. As a result, most of the software lack or partially equipped with documentation.

The same dilemma exits in the software architecture field. It has been shown in several studies (e.g. [2, 4, 10]) and as discussed in section 1.2 of this chapter that one of the most common reason of software architecture erosion is the poor or complete lack of documentation.

It has been shown in several studies (e.g. [1, 15, 18]) that only few software development organizations actually manage to keep the documentation of the software architecture updated. Most of the organizations start documentation at the beginning of the project but as the project proceeds, the documentation becomes of low priority. As a result of this, the documentation gets outdated and is no longer useful for the organization. In many cases there is no single source of documentation to consult and at the end source code is left as the only up-to-date documentation of the system.
CHAPTER 2: Vision Based Architecture Method (V-BAM)

In this chapter we present a method to develop the software architecture of the next release (N+1) on the basis of an architecture vision, an architecture design document and the architecture of previous (N) release. We named this method as Vision-Based Architecture Method (V-BAM). The V-BAM method provides a disciplined approach to plan the software architecture development activities. Its goal is to ensure the production and evolution of the high-quality software that meets the needs of its end-users within a predictable schedule and that reduces or avoid architecture erosion. V-BAM facilitates the development of the software architecture by providing the concept of architecture vision, synchronization of the architecture with the implementation, maintaining architecture design decision and goals of each release separately and encourages performing software architecture related activities during each release.

2.1 Goals of V-BAM

The basic goals of V-BAM are as follows:

1. Place the architecture vision idea into practice by implementing a systematic iterative architecture development process.
2. Assist in developing and maintaining high quality software architecture throughout the life of a software system.
3. Structure the efforts required to implement the next release of a software system.
4. Reduce and overcome architecture erosion and drift along with their related problems in order to extend software architecture’s (and software’s) lifespan.

2.2 Pre-requisites

The software architecture is mainly a reflection of the software requirements. It is important to know what is required to develop before actually design the architecture. The pre-requisite of V-BAM method are:

- Functional and quality requirements of the system.
- In addition business rules and constrains should also be identified.

If these pre-requisites are not satisfied then it can affect negatively the architecture vision and architecture of next release of software system. More importantly if vision if developed initially on bases of partial or incorrect requirements then it would not be much helpful to guide the software architecture.

2.3 Input / Output of the V-BAM

Software requirement specifications are mostly developed in the requirement engineering phase of the software development life cycle. This process assumes that the software requirements are analyzed, elicited and documented. Software requirement specification is the main input to V-BAM process at the start of the project for the first cycle of the V-BAM process. Architecture vision, Architecture design document and software architecture for the current releases are developed in the first cycle of the V-BAM method. These documents along with the requirement specification document serves as input for the subsequent cycles of the V-BAM method as shown in the figure 2.1.
Figure 2.1 – Vision based process, its inputs and output

The figure 2.1 explains the inputs and output of V-BAM. The V-BAM has four inputs and output. In general V-BAM takes requirements specification, architecture vision, architecture design document of previous release (is exists) and release specific architecture of previous release. The output of V-BAM is release specific architecture of current release.

2.4 V-BAM Activities

The V-BAM method consists of the following five steps as shown in the figure 2.2. These steps are sequential and they occur in the same sequence for the each cycle. These steps are then explained separately for the purpose of clarity and understanding. In reality and practical life these steps pretty much overlap each other and it is hard to draw a line between them.

1. Collect / identify architecture requirements
2. Develop/refine the architecture vision
3. Develop release specific architecture design document
4. Develop release specific architecture.
5. Synchronize release specific architecture with vision and architecture design document
Step 1: Collect/Identify Architecture Requirements
The requirements describe what the software should do and in a technical perspective the software architecture is mainly a reflection of the software requirements. It is important to know what is required to develop before actually design the architecture. There are a number of studies [25, 26, 28] that show how to identify the technical business requirement for the software architecture.

The main purpose or goal of this step is to analyze and identify the architecture requirements, the requirements that can affect the software architecture (at any level, to any extent). Following activities can be performed or can be considered during this step.

- Both functional and quality requirements of the system should be analyzed carefully to identify architecture level requirements.
- The constraints at business, domain and technical level should be analyzed closely as they mostly lead to the architecture requirements.
- The business rules and polices of the targeted domain should be considered for the potential architecture requirements.
- Special quality requirements (*abilities) of the system should be considered more deeply as they in most cases lead to the architecture requirements.
The software requirement specification is the main input to this step and the identified architecture requirements are the main output. In addition to the software requirement specification there can be other inputs to this steps like business rules, constraints etc, as shown in the figure 2.3. These architecture requirements form the basis on which the proposed software architecture is evaluated in term of its ability to fulfill business rules, constraints, functional and quality requirements etc.

This step starts after the requirement engineering process that works as input to this step where the requirements are identified, analyzed and elicited for the software application. This step is finished when the architecture requirements are identified and gets separated from the others system requirements.

**Step 2: Develop/Refine the Architecture Vision**

Architecture Vision is a document that contains the overall goals and objective of the software system. The architecture vision represents an ideal architecture of the system that satisfies the functional and quality requirements in an ideal way. This ideal picture serves as a goal to inspire. In addition it provides the guidelines and motivation for the architecture in next release(s). It is not important that whether it is actually possible to achieve these goals currently but it is more important to have some ideal goals to strive for. As shown in the figure 2.4 the architecture vision mainly acts as a specification of the system and the architecture design document is an instantiation of this vision. The goals, objectives etc are defined in the architecture vision, and are implemented in different releases of the system.

Van Gurp & Bosch identified that very few software development organization actually manage to keep the updated version of the architecture documentation [32]. It is a general practice that the architecture document contains a mixture of the
architectural documentation and documentation of goals, objectives etc [1]. This leads to a situation where the architecture documentation contains a history of design decisions, constraints etc including those design decisions that are expired [1]. To address this problem V-BAM propose that goals and objectives should be documented in a separate file. In V-BAM, the software architecture document is split into two separate documents, the architectural design document of the current release and the architecture vision as shown in figure 2.5. The software architecture is recorded in one document and another document contains architecture goals, objects, constrains, etc. The architecture vision is mainly developed at the application level and the architecture design document is developed at the release level.

Figure 2.5 – Division of the traditional architecture document into two documents

**Architecture Vision Creation/Refinement:**
In this section we discuss how to actually create the architecture vision, what things that are important to consider, what are the steps we should follow to create the vision? There are number of factors that can affect the architecture vision and needs to be considered. Here we are going to discuss few factors that are important from our perspective; other factors depend on the particular organization and its properties. We have structured these factors in the order of importance starting from the most important and moving toward the optional factors.

**Goals and objectives:**
The architecture vision should be created by having the business and technical goals in mind. The details of the application architecture goals and the objectives to meet those goals should also be documented here. In the architecture vision both the short term and long term goals should be documented. The presence of these goals and objectives in the architecture vision guide the development of each release and play important role in success of software application.

**Design decision:**
Architecture design decisions should be documented in the architecture vision. It helps to structure and manage the architecture design decisions throughout the life cycle of the software system. While designing the release specific architecture design documents, these design existing design decisions in the architecture vision will serve as a guideline.
Architecture constraints:
The constraints at architecture level that need to be fulfilled during the development, maintenance and management of the system architecture, should be documented in the architecture vision. Usually the constraints are defined at different levels like enterprise level, project specific constraints (like resources, time, money) etc. The presence of these architecture constraints in the architecture vision ensures that the software architecture should not violate any of them in any release.

Architecture rules:
The architecture rules should also be document in the architecture vision. These rules serve as the guidelines during the evolution and maintenance of software system.

Frameworks and principles:
The architecture vision should contain the targeted frameworks along with the architecture principles that is planned to be followed during the development of the software architecture and development cycle. It assists release architectures to follow these frameworks and principles in a consistent way by getting inspiration from the architecture vision.

During the development/evolution of the software system, if any requirement has effect on goals and objectives and other factors that are defined in the architecture vision then we need to change the vision. Following are the steps to introduce changes in the architecture vision.

1. Analyze the impact of the requirement on the architecture vision: Analyze the requirements to see if the requirement has any impact on any factor that is defined in the architecture vision. For example if the requirement affects the goal and objectives of the system then we need to refine the architecture vision.
2. Impact analysis of the change: If the new requirement affects the architecture vision then identify the impact of the change.
3. Apply changes in the architecture vision: As analyzed in the pervious steps if the new requirement affects the architecture vision then we need to change the identified areas of the architecture vision.

This step starts when the architecture requirements are identified that works as an input to this step. This step is finished when the architecture vision is developed for the software application. The output of this step is an updated software architecture vision document that reflects the current and future architecture of the software system.

The goal of this step is to build or refine a software architecture vision. One important thing to note is that this architecture vision is developed and refined at the software application level and not for any particular release. The major input to this step is the architecture requirements identified in the previous step. Apart from the requirements, business rules, architecture style, technical & business constraints, resources, strategies etc. also act as input to this step. From the second release and onward the architecture design document and software architecture of previous release are also used as input.

**Step 3: Develop Release Specific Architecture Design Document**
Now at this point, we have the architecture requirements, an ideal picture in the form of an architecture vision and release specific architecture requirements. In case of second release and onwards we also have architecture design document and software architecture of previous release as an input to this step. In this next step, we narrow down our focus and concentrate on a specific release and develop an architecture design document for the release at hand.

The main goal of this step is to develop an architecture design document of a specific release where total focus is given to release ahead within given resources and time. The inputs of this step are the architecture vision, architecture of the previous release (if it exists) and the release specific architecture requirements. The output of this step is a release specific architecture design document. This step starts when the architecture vision is established and the architecture requirements are identified for a specific release. This step is finished when the architecture design document is written. The design document defines the path on which the software architecture needs to build in next step. Following activities can be performed or can be considered during this step.

- It helps to define release specific boundaries and constraints, what is included and what is not included in the release [33].
- Identify the areas in the architecture of the software that are affected by new requirements. Analyze effects of new requirements on the rest of the system and take appropriate action for this.
- While packaging the requirements for specific release it is important to consider available resources for that release.
- Here we basically record the method that will be implemented to develop the software architecture.
- This step includes efforts to make sure that the release specific architecture follows those standards and styles etc that are specified in the architecture vision or tries to keep it close to the architecture vision.
- The Requirements are analyzed in details and decision is made about what should be developed and how to develop it.
- Document the architecture design decisions and the rationale for these decisions so that it is maintainable and understandable in the future.
- Document different views at different abstract level to show different interesting aspect of software architecture.

**Step 4: Develop Release Specific Architecture**

The main goal of this step is to build the software architecture for a specific release. In this step the architecture that was implemented in the previous release is evolves into the one prescribed by the release specific architecture documentation. The release specific software architecture is in a concrete form that guides the development of the software application for the specific release. It is the realized form of the architecture design document and inspired from the architecture vision. The inputs to this step are the architecture specific requirement package and the software architecture design documents. In case of second release and onwards we also have architecture design document and software architecture of previous release as an input to this step. The output of this step is the software architecture of a specific release. This step starts after the development of the software
architecture design document. The step is completed when the software architecture is completely updated.

- The software architecture (build in this step) is realized form of the abstract architecture and architecture design document.
- During software architecture development, focus on the development of the core part of software architecture that lead to the development of specific release.
- The main computational and data components of the system are identified.
- Dependencies and relationship among the components is identified.
- The interaction between the components is defined.
- The specific architecture style, architecture pattern and principles should be implemented that is specified in the vision or release specific architecture design document.
- The architecture decisions, constrains and business rules should be implemented that are specified in the vision or release specific architecture design document.
- Keep track of deviations from the release specific architecture documentation of the previous release.

**Step 5: Synchronize release specific architecture with vision and architecture design document**

At this point, we are done with software implementation that followed the architecture vision, software architecture design documents and software architecture. Although these steps and artifacts follows each other there is a possibility that they differ at some points because of different reasons like time constraints, changes in the requirements at runtime etc. The goal of this step is to synchronize the software implementation with the software architecture, software design document and architecture vision. The synchronization helps to keep these artifacts aligned and up to date, that helps to overcome the architecture erosion and drift.

This step starts at the end of the each iteration when architecture is developed and software is built around it by implementing the software architecture. This is the last step of this process that is finished when the architecture vision, the architecture design document, the software architecture and its implementation are synchronized. To synchronize these artifacts, identify the inconsistencies at different level like in the implementation, software architectures, architecture design document and architecture vision. Once inconsistencies are identified update the relevant documents to accommodate these inconsistencies. The inputs of this step are software architecture vision, architecture design document and software architecture. The outputs of this step are a synchronized version of the software architecture vision, architecture design document and software architecture document.

**Process of Synchronization:**
In the process of synchronization, the implemented architecture should be placed at the center location. Implemented software architecture cannot be changed for current release as it is already built and delivered. In the synchronization step, the
architecture design document and architecture vision are updated accordingly. Following are main steps in synchronization.

- Find inconsistencies among the implemented architecture and architecture design document.
- Update architecture design document accordingly. The architecture design document is updated so it depicts the real picture of software architecture that is implemented in current release.
- Figure out the requirements (if any) introduced into the system at later stages of development life cycle.
- Find out the effect of requirement (identified in previous step) on architecture vision.
- Update architecture vision according so that vision should represent latest ideal picture of the system.

Note: The creation and updating of architecture vision is discussed in section 2.3.

**Measurement of deviation:**
To provide a comprehensive method/process to measure the architecture deviation from vision is not part of our thesis, as it is complete study by itself. But here we are providing a kind of checklist that will be helpful in measuring the architecture deviation from architecture vision.

- The measurement can be considered at component, module or class level depends on effect of vision that can be variant.
- Select a goal in the vision for which we want to calculate deviation.
- If the selected goal of vision has effect on class level then count number of classes that has been updated to achieve this goal. For example, we have a goal in vision that each class should be initialized through configuration. So, calculate total number of classes in one module and calculate the total number of classes that have been initialized by configuration during current release. The difference will show that how much far away we are from our goal written in vision. Deviation = Total number of classes – config initialization class
- If the selected goal of vision has effects on module/package level then count number of classes that has been updated to achieve this goal. For example, we have a goal in vision that facade pattern should be used among the components. So, calculate total number of components that have been interacting using facade pattern during current release. The difference will show that how much far away we are from your goal written in vision. Deviation = Total number of component – component interacting using facade

### 2.5 Summary
The V-BAM method is a systematic way to overcome/reduce software architecture erosion. It puts focus on the creation of the ideal picture (architecture vision) of the software architecture and mapping this ideal picture with implemented architecture developed in previous release so that we have an software architecture for next release (N+1). The major inputs of V-BAM method are software requirement specification, architecture vision, architecture design document, release specific architecture of next release (n+1). The V-BAM method consists of the five steps. In
first step both functional and quality requirements of the system are analyzed carefully to identify architecture level requirements. In the second step the architecture vision is develop/refined. The architecture vision represents an ideal architecture of the system that satisfies the functional and quality requirements in an ideal way. This ideal picture serves as a goal to inspire. It is not important that whether it is actually possible to achieve these goals right now but it is more important to have some ideal goals to strive for. Once we have architecture requirements and architecture vision, in next step release specific architecture design document developed where main focus is given to current release within given resources and time. This design document defines the path on which the software architecture needs to build in next step. In the next step release specific architecture is developed. The release specific software architecture is in a concrete form that guides the development of the software application for the specific release. It is the realized form of the architecture design document and inspired from the architecture vision. In the final step of the method we synchronize release specific architecture with the architecture vision and design document. The main intention of this step is to synchronize the software implementation with the software architecture, software design document and architecture vision. The synchronization helps to keep these artifacts aligned and up to date, that helps to overcome the architecture erosion and drift.
Chapter 3: V-BAM Validation

In this chapter, we present a validation of our proposed method ‘V-BAM’. To perform the validation of the V-BAM method we have selected an industrial software product and execute all steps of the V-BAM method to study the implications and benefits of the V-BAM method.

3.1 Company

For our validation we have selected a Stockholm based company that is working in software and mobile marketing industry. There are around 30 people working in the company under different roles in different projects. We are unable to expose the name of our targeted company due to confidentiality reasons. In this study we will use the name ‘the company’ for our targeted company, instead of using its original name. The main reasons behind the selection of the company are that the company is willing to participate in the case study and most of the relevant information is accessible.

3.2 Product

The name of the selected product is ‘Content Ingestion System’ (CIS). This product is in the domain of media and Content Management Systems (CMS). CIS is a specialized system for processing of content deliveries. The content deliveries are mainly musical album/s and/or track/s. The content deliveries consist of metadata and media (audio binaries) files in various formats. The metadata is in the XML format that contains information about the binary contents in the deliveries.

The main purpose of the CIS is to provide a link between the content providers and the ‘content delivery platforms’. CIS takes contents from the content suppliers as an input, performs some validations and other checks and transform the musical deliveries into different formats. In CIS the transformation of musical deliveries into the different formats is termed transcoding. Once transcoding is performed CIS publishes the albums or tracks to the publishers according to defined business rules and rights. Moreover, CIS can also export the contents to different media delivery platforms.

This product has existed in the market for quite some time and has multiple releases. The product is neither very small nor very large, it is a medium scale product. Many architectural and design level changes have occurred, during the multiple releases of the product.

3.3 Method of Getting Data

To study the architecture of the product we have used the following techniques to collect the required information and data.

- Interviews with the developers and managers.
- Studies of relevant documents like requirement specifications, standards, design document, change logs, user manuals, etc.
- Studies of source code of the product.

3.4 Problems in the CIS Product

In section 1.2 of chapter 1 named “Software Architecture Erosion” we have identified the causes and effect of the software architecture erosion. In CIS we have found a number of problems that refers to the problems identified in the chapter 1. These
problems clearly show the symptoms of the architecture erosion in the CIS product. Problems in the CIS are explained below in detail:

The architecture and design of the CIS is not developed and maintained explicitly:
The architecture and design documents are developed in the beginning of the project but they are not updated on the regular basis. There is no one in the team that is solely responsible for architecture and design tasks. We have experienced that from release 1.6 onwards most of the requirements were incorporated directly into the code and there are no corresponding changes in the architecture and design documents. Source code is thus the only available updated document. The CIS product is dependent on the few key persons who have been involved in the project from the beginning. The design decisions of CIS lacks clarity and coherence, it is hard to track design decisions from document and code. Since there was no rationale for the design decisions it became hard to follow the original architecture.

Architecture style is not consistent among different releases:
For example, the ‘three tier’ architecture style was decided to be implemented in CIS, but after some releases the three tiers (presentation tier, business logic, Database access) were implemented either poorly (these tiers are not cohesive and contains extra or unrelated code or logic) or not followed at all. For example, at some points business logic and business decisions were embedded in the ‘database access’ tier. In addition, at some points the business tier contains the logic of database access. The GUI and publishing components were mainly affected by this problem.

Implementation is not aligned with the design and architecture:
In CIS, at many points the implementation is not aligned with the design and the architecture. For example, it was decided that ‘Aspect Oriented Programming’ and ‘Ingestion of Control’ (offered by Spring 3.0 in Java) shall be used. This design decision was followed properly during the initial releases but as the system evolved the implementation started to deviate from the original design. So after some releases the source code contained a mixture of inversion of control and self creating objects. There were different reasons for this deviation like lack of knowledge, lack of proper training, time constraints, new developers joined the team and no code or design review etc. Moreover design level aspects were embedded into the code and were not defined properly.

Over the period of time CIS became complex and hard to understand:
After some releases, the CIS product became complex and relatively hard to understand. A number of factors play a role in increasing the complexity of the CIS product. Different classes and components are more coupled and less cohesive. One class or component was performing many tasks, one tasks is scattered among a number of classes that made it difficult to track and understand. The changes in the existing code became very expensive and hard to incorporate because of high dependency among different classes and components.

Redundancy of code:
A lot of the common functionalities already exist in different common libraries but many (new and some old) developers tends to develop their own implementation because they were unaware of existing common functionalities. Almost each sub-system has a common package with the name ‘util’ that contains a bunch of classes. The basic reason behind this situation is lack of training, less or poor communication within the team.
Re-factoring of the code and design is not performed on a regular basis: This condition leads to a situation where the implementation contains unused and/or deprecated code, and classes and functions are not divided properly. The code in different classes and functions overlaps and is redundant. This situation causes high coupling and low cohesion among the classes and components of the system that leads to an increase in complexity. For example ClientEngine in GUI Server and Validator Components of ImportServer have high coupling and functionality is scattered over different classes. It was hard to define the boundary of these components. The Import component of ImportServer contains plenty of unused code, this code is not deleted just in case it might be in used somewhere and deletion may cause some bugs in the existing implementation.

Business rules are merged with the requirements, they are not documented separately: The business rules are not documented separately. They are merged with the software requirements. This leads to a situation where it is hard to track the business rules and to see dependencies among them. There are lots of business rules in the Import and Publish components but it was hard for us to track and list them separately. Moreover, documents contain many outdated and deprecated business rules. This situation further increases the complexity of the system.

The problems in CIS are quite similar to the problems of erosion that are identified in the section 1.2 of chapter 1. Here we have summarized how V-BAM method helps to overcome these problems. In CIS architecture and design was not developed and maintained explicitly. Since V-BAM focus on having a release specific design document that encourages to develop and document architecture explicitly for each release. Moreover V-BAM separates the architecture design decision from the software architecture that helps to track design decision as they are documented separately. Another problem in CIS was that the business rules were merged with the requirements and there were inconsistency/violation of the architecture style. In V-BAM architectural style that is adopted for the system is documented in the Vision and each release gets the inspiration from the vision. It helps to keep business rules separated and the architectural style remains consistent in each release. Moreover after each release synchronizations step is performed to make sure that the implementation is kept aligned with the architecture vision. In CIS there was another major problem that implementation was not aligned with the documented design and architecture that made it complex and hard to understand. The main focus of synchronization step in V-BAM is to align implementation with release specific architecture and architecture vision. In addition the iterative nature of V-BAM accommodates the discrepancies (identified in the synchronization step) in the next iteration.

### 3.5 Application of V-BAM

**V-BAM Implementation**

In this section we execute the V-BAM method on the CIS product and analyze the implications and advantages of the V-BAM method. Just for the sake of clarity we have combined the implementation of the V-BAM method for all releases. Although in practice the V-BAM method will be implemented separately on each release.

**Step1. Identify architecture requirements**
In the CIS product the architecture level requirements were not explicitly documented. We have received the requirement specification, code, release notes and change logs for each release from the company. We have used these documents to extract the architecture level requirements of the CIS product for each release. The identified architectural requirements (functional and quality) of release 0.5, 1.2 and 1.5 are given in appendix B.

**Step2. Develop/Refine architecture vision**
We have developed the architecture vision of the CIS product by considering the company possible strategies, current and future requirements. Here we have tried to consider the ideal representation of the application and took some assumptions about the future of the CIS product. The architecture vision for CIS product is listed in the appendix A.

**Map Requirement to vision**

**Release 1**

**Requirement:**
- The CIS product should be able to import musical deliveries in different formats from different known and unknown suppliers.
- For each track transcoding should be performed within 5 min.
- Each issue should be resolved within 5 sec after decision is taken by the user.
- System should be able to export albums (or track etc) to different media delivery platforms.

**Vision:**
- Different design patterns will be used to solve generic problem in a standard way. More specifically at Imports, transcoding and publish modules the standard Structural patterns (Adapter, Bridge, Decorator, Facade, Flyweight, Proxy) will be used depending on situation.
- In long term, system should be able to ingest games and other application content in parallel to music contents.
- The overall system should be very fast. The album or track should be ingested and published within minimum time period.
- The architecture should be structured in a way to achieve high performance.

**Release 2:**

**Requirement:**
- It should be possible to perform transcoding on the multiple machines.
- It should not be possible to ingest, publish and export album without CIS user permission.
- Transcoding should be performed in the secure environment.

**Vision:**
- Different components of the system should run in distributed environment.
- Application usage and communication should be highly secured. Different security measures should be used like roles and right, secure certificate etc.
- Different components should interact using secure protocol and should use different encryption techniques to encrypt and decrypt data.

**Release 3:**

**Requirement:**
- The publish component should run as a separate service.
- The publishing component should be configurable.
- The export component should be configurable.

**Vision:**
- One consistent architecture style should be used through out the application. In CIS we will use service oriented architecture.
Dependences between different components should be defined through the configuration files. Dependency logic should not be embedded in the components. CIS architecture and design should follow the idea of Dependency Injection.

Step3. Develop release specific architecture design document
We have extracted the architecture requirements for all three releases from the existing system. Different sub-system, components and interaction among them are introduced into the system based on these requirements. The transformation of the requirements of each release on the architecture of the system is presented below:

Release 0.5:
At conceptual level we have decomposed the CIS product in three sub-systems. The decomposition of the system is performed on the basis of the requirements for the release 0.5 (defined in the step 1) and the architecture vision. As defined in the architecture vision we have applied different design patterns and architectural styles to reduce the dependencies among different sub-systems. At sub-system level we have used Façade design pattern to provide the single interface for the interaction among different sub-system. These sub-systems are further composed of several components. The detailed description of these three sub-systems and components and the requirements they are satisfying, are presented in appendix B.

Release 1.2:
The decomposition of the system is performed on the basis of the architecture requirements for the release 1.2 (defined in the step 1) and the architecture vision. In this release architectural requirement does not have major impact on the overall architecture and design of the system. Most of the architecture level requirements in this release are related to the TranscodingServer sub-system. We have added/changed or replace some components based on new architectural requirements but overall picture remains the same. The detailed description of sub-systems and components and the requirements they are satisfying, are presented in appendix B.

Release 3.2:
The decomposition of the system is performed on the basis of the architecture requirements for the release 3.2 (defined in the step 1) and the architecture vision. This release had relatively major impact on the architecture of the system. At the conceptual level BatchServer sub-system is split into four different sub-systems that are ImportServer, PublishServer, ExportServer and Core. There are several reasons to split the BatchServer sub-system. As a result of new requirements, the complexity of BatchServer was increasing due to the increased coupling between the components. Service oriented (SOA) architecture is used to reduce the coupling between the sub-systems and make them easy to manage in the future. Moreover the introduction of SOA takes the architecture closer to the vision of the system. The detailed description of sub-systems and components and the requirements they are satisfying, are presented in appendix B.

Step4. Develop release specific architecture
During this step we developed the architecture of the system. The architecture of the system was developed on the basis of architecture vision and architecture design document. For each release the conceptual level architecture diagrams and its rational are given in appendix B.
Step 5. Synchronize architecture with vision and architecture design document

In this step we have analyzed the architecture of the previous release to find any deviation from the architecture that was supposed to be implemented. Once we have identified the deviation we have compared the implemented architecture with the architecture vision. On the basis of these findings we have designed the architecture for the next release in a way that will reduce the gap between the implemented architecture and architecture vision.

Here we present our findings from applying the V-BAM method on the CIS product. Like any other method the V-BAM method has number of advantages and disadvantages. The advantages of V-BAM methods tend to show after the initial releases. The advantages and disadvantages of the V-BAM method are discussed below in detail:

3.6 Experiences from Applying the V-BAM Method

The V-BAM method provides a systematic way to develop and maintain the software architecture based on the architecture vision. We have observed a number of interesting things while performing the validation of the V-BAM method. First of all, the vision creation step was not that easy to perform as we have to keep room for future changes at the same time we have to maintain the balance between the ideal architecture and the architecture that can be implemented.

Furthermore, when we were planning the Release 0.5, it appears to be some extra activities to perform as there were not many complexities in that release (as discussed in the section 4.6).

It took quite a while to see the real benefits of the V-BAM method. The iterative nature and vision based approach of V-BAM provides a clear benefit over the traditional method used in CIS. In the traditional method the direction of the architecture is unknown; there is no idealized architecture to relate. Every time there is some new requirements, architecture of the system is molded to facilitate that requirement. On the other hand in V-BAM method after each release the implemented architecture synchronized with the vision and release specific architecture. It helps us to keep the architecture up to date and consistent with the implementation thus reducing the architecture erosion.

The implementation of the V-BAM in three releases just provides the glimpse of the advantages over the traditional methods but there are still quite a lot of factors that needs to be explored.

3.6.1 Advantages

Increased durability, soundness and robustness:
The durability, soundness and robustness were affected because of many reasons (identified in section 3.4) like the architecture style is not consistent among different releases, redundancy of code, refactoring of the code and design is not performed on a regular basis, random change.

The V-BAM method addresses these problems to increase system durability, soundness and robustness. The core of the V-BAM method is to synchronize the

implemented architecture with the architecture vision during each release that helps to detect architecture erosion at the early stages.

In CIS when we were designing the vision we foresaw changes in the requirements related to Transcoding. We got inspiration from the architecture vision and developed the Transcoding as a separate sub-system. That makes it easy to incorporate changes in the Transcoding system when new requirements are added in the release 1.2 and release 3.2, the details are discussed in Appendix B. Furthermore we got inspiration from the software architecture vision and adopt same architecture style from the beginning of the project and verify consistency of architecture style during the synchronization step of each release.

**Traceability of design decisions:**
In the CIS system the design decisions were not traceable due to some reasons (identified in section 3.4) like the architecture and design of the CIS is not developed and maintained explicitly, implementation is not aligned with the design and architecture, business rules are merged with the requirements and are not documented separately.

When we studied CIS system, the software architecture and design of the system was not explicitly documented. There were few architecture documents that explain the software architecture of CIS to some extent but they were out dated. To get more understanding of the system we had to study code and file structure the CIS system.

The V-BAM method addresses the problem of traceability in many different ways. The V-BAM method is based of software architecture documentation and it encourages to develop separate documents of architecture vision, software architecture design document as discussed in chapter 2. The V-BAM facilitates the traceability of design decision as it starts with identification of architecture requirements. These architecture requirements are later used to create architecture vision and architecture design document for each release. Since V-BAM encourages managing the design decisions as a separate document that makes it easy to track existing or old decisions and update them according to new requirements. Moreover it also helped us to detect and analyze the impact of changes.

**Reduce and manage the complexity of the system:**
The CIS system was complex because of many reasons (identified in section 3.4) like the architecture style was not consistent among different releases, implementation is not aligned with the design and architecture, redundancy of code.

As stated earlier, in the CIS system the software architecture and design of the system was not updated to reflect the latest changes that made it hard to make changes in the CIS system on latter stages. The changes were directly incorporated into the code without considering its impact on the overall architecture of the system. Moreover when we studied the code of CIS system we found redundant code in different components that were developed in different time frame.

The focus on the documentation, vision centric and iterative nature of the V-BAM method helps to reduce the complexity of the system. During each release of CIS, we have explicitly developed the software architecture of each release. In release 0.5 of CIS we have developed the architecture vision in which we have documented goals and objective, design decision, system constrain etc of CIS system (for detail
see Appendix A). The existence of vision in CIS simplifies our release specific design document as it only focuses on release specific details. In addition, at end of the each iteration we have performed the synchronization step to align the implementation with the release specific architecture and the architecture vision. These activities performed in V-BAM helped us to make the software architecture consistence and reduce the overall complexity of the system.

**Release pace:**
The release pace of the CIS system was affected as it became hard to understand and the complexity of the CIS system is grow, at the end of release 3.2 the CIS system gets eroded (for details see Appendix B).

The V-BAM method emphasize on putting the efforts on development and management of the architecture during each release. It is hard to say that V-BAM have any significant effect the release pace of the CIS system as we have performed only three iterations on existing system. What we have noticed during our limited experience with CIS that it was not eroded at the end of release 3.2 in contrast to previous implementation of CIS. On the basis of this we can say that in the future releases the release pace might increase or remain the same as we put regular efforts on architecture during each release.

**Method based Approach:**
It is better to have a method to support rather than not having any method as it drives the development and keep track of important things to be performed. A method is a systematic procedure and guidance on how to deploy one or more notations for describing a problem or solution domain [24]. It is of secondary importance that how good or bad a specific method is, what more important is to at least have some guidelines to follow. The V-BAM method provides a systematic way to facilitate the participants to think broadly about their requirements and needs (what they want to do). The V-BAM method contains the steps that encourage user to have architecture vision based on requirements and it requires having an ideal representation of the system and provides a way to reduce the gap between the ideal representation and actual implementation.

### 3.6.2 Disadvantages / Restriction

**Extra Effort:**
The V-BAM method requires some extra effort as you have to create the architecture vision, develop and maintain a separate document for the software architecture and architecture design decisions, synchronize vision with implemented architecture, etc. During the release 0.5 and release 1.2 of CIS there were not many changes in the requirements and design. To perform the steps of the V-BAM method appears to be more effort consuming as we have not seen the real benefit of the V-BAM method yet.

**Time Consuming:**
The V-BAM method is a bit more time consuming as it requires some steps to be performed after each release. We have to take care of many aspects like future requirements, market trends, business and technical goals etc.

### 3.7 Scalability of V-BAM
The focus of V-BAM method is to minimize the architecture erosion that occurred due to the software evolution. So the activities of V-BAM method in itself are developed keeping the iterative nature of the project in mind. The duration of each release or iteration can affect the application of V-BAM and its result. The V-BAM method requires putting time on development and refinement of the software architecture and vision on each release. In addition, V-BAM demands to put some time on software architecture development/refinement and on synchronization step. In the project with frequent release these steps could cumbersome as one has to perform all activities quite frequently. On the other hand if there is the probability that the requirement of the project is changed quite frequently, it is better to have frequent iteration because that will enable to refine the architecture vision and synchronize architecture design documents. It is the trade off and it mainly depends on the nature of the project.

In a small scale project, V-BAM method might not be of much help. It does not mean that the V-BAM method is not implement able on small scale project, but it is the fact that small scale project might not reach the point where you can see the real benefits of the V-BAM method. Normally in small scale projects, the software architecture is not much big, complexity level is not much high and project goals and objectives are not much longer term. Although we can use V-BAM method for small scale project but it might not give the desired results.

We have validated V-BAM method on a medium size project. On the basis of our experience presented in the section 3.6 we can say that the V-BAM method best suited for the medium and large scale project. It would take a while to see the real benefits of the V-BAM method as in the beginning it requires more efforts to build the architecture vision and architecture design document. Once we have these documents in hand we can relate changes in the requirement with them. Step 3, 4, and 5 of V-BAM encourages to document design decisions as separate document that makes it easy to track decision decisions. V-BAM helps to keep the architecture up to date and consistent with the implementation thus reducing the architecture erosion.

For the very large scale project, V-BAM method is still productive as in case of medium to large scale project but it might take bit more time. In a very large scale projects it is more likely that one has to refine the architecture vision itself as the requirements of the software evolve due to change in the environment (changes in hardware, software, technology, market needs etc). Due to the increased probability that architecture vision is changed the steps to develop release specific architecture design document and release specific architecture will take bit more time.

### 3.8 Validity Threats

In this section we discuss potential validity threats to the validation of the V-BAM method.

**Conclusion validity**

Conclusion validity threat is relevant whenever we are trying to decide if there is a relationship in our observations. Threats to conclusion validity can lead us to either find a relationship where it does not exist or to miss the relationship where it exists.

The selected company and project used for the static validation can pose a threat to validity, since projects are different in size and type. To minimize this threat we have selected the company that has well versed experience in the field of software
development. Moreover the project that we have selected is of medium size and is in the market for quite some time.

**Internal validity**

*Selection Validity:*  
In order to study CIS system, we have conducted discussion/interviews with members of development and management team. It is threat to our validation that some of the selected people does not have complete picture of the CIS system. To overcome this threat we have selected members that have been working the system for quite some time. Moreover we have selected members from different department to get different perspectives of the system

*Additive Effects with Selection:*  
For validation, we have performed interviews and emails correspondence. It is quite possible that the interviewees have responded restrictively because they are conscious of the fact that their opinion was recorded. In order to overcome this threat, anonymity of the interviewees was guaranteed. However, in a limited time it was not possible completely eliminate this threat.

**External validity**  
External validity is related to the generalization of study, i.e. the degree to which our study would hold for other persons into other places during other times.

*Project Type Validity:*  
There are several different kinds of software projects with different domains and nature. We have performed the validation of the V-BAM method on only one type of project that puts question mark on the validity of the research. Since different phases of software development life cycle are quite independent of the nature of the project and moreover V-BAM is quite customizable, so we can still maintain the validity of the research.

*Time Validity:*  
The time when we have performed the validation of the V-BAM method on the CIS product can also affect the results. It is observed that architecture of the software erode over the period of time. To maintain the validity of the study we have selected several different releases of the software product starting from the initial release to the most recent release.
Chapter 4: Discussion

The V-BAM method facilitates the participants to think broadly about their requirements and needs (what they want to do). The vision creation step of this method encourages to think without much considering the constraints like time, resources etc. The other steps of the method help to implement the real possible architectural concepts in multiple iterations under the given possible constraints. The overall idea of this method is to have activities that help to create an ideal architecture of the system along with activities that helps to achieve this ideal architecture while considering limitations. So that the overall architecture remains align with the architecture vision that will help to avoid the architecture erosion and its related problems. Once the vision is established, this method helps and guides each of the following releases to get closer and keep closer to the vision. Under dynamic and changing environment, the vision can be updated if required.

The V-BAM is a configurable process. No single process is suitable for all software development. The V-BAM method fits medium scale projects as well as large scale projects. The V-BAM is founded on simple and clear architecture process and it can be varied to accommodate different situations. The V-BAM method captures many best practices in modern software development in a form that makes it suitable for a wide range of projects and organizations.

In this study we have not discussed any special software development methodology or process like RUP, agile, water fall etc. In addition this method does not conflicts or provide alternative of any of these development or management process or method, it exists in parallel to them. Any software development or management method can be adopted. This method lies in ‘architecture development and maintenance’ phase and suggests the way in which software architecture should be developed and managed successfully.

The implication of VBAM does not get affected by the method used for development software system and life span of the product. It is not a replacement of any software development method. It is basically a method used to develop/maintain software architecture and lies in architecture development stage of software development life cycle. It provides an iterative and discipline approach to develop software architecture of next release while avoiding the architecture erosion. It is very common practices that software is not developed at once; instead it is developed in multiple releases. The requirements of the system continue to change or added and as a result the software architecture evolves to cater these changes. The architecture vision guides this architecture evolution process. The VBAM method provides a systematic way to update the software architecture continuously during each release and synchronize architecture with design document and architecture vision.
In the following section we have discussed how we have answered our research questions in this study.

**How can erosion affect the architecture of evolving software?**
Architecture erosion is defined as a phenomenon in which an application’s initial architecture is modified arbitrarily to the point where it no longer holds its key properties. The architecture of the software moves away from its original shape if the architecture is changed without taking into account the impact of the change. Table 1.1 in chapter 1 summarizes the list of problems caused by the architecture erosion. Few major problems caused by the architecture erosion are inadaptability of architecture as it lost focus and clarity, traceability of the requirement, hard to determine requirement change impact and maintainability of the system.

**What is the current state of research to overcome architecture erosion?**
Section 1.2 of chapter 1 presents some existing studies/efforts that have been conducted to analyze and overcome the causes of architecture erosion. From the existing research, it is observed that the lacking or outdated architecture documentation is the major cause of the architecture erosion. A number of studies have been done to overcome or reduce the problem caused by architecture erosion. Most of the existing studies emphasize documenting different artifacts of software development.

**What is the architecture vision, how would it help to reduce or limit architecture erosion?**
This question is mainly addressed in section 2.3 of chapter 2 in detail. The architecture vision is the document that represents overall goals of the software system at a high level of granularity. The architecture vision represents an ideal architecture of the system that satisfies the functional and quality requirements in an ideal way. This ideal picture serves as a goal to inspire. The architecture design document represents an instance of the vision for a particular development release. It serves as a guide to which changes can be related to that helps to avoid any pitfall that can lead to problems of the architecture erosion and architecture drift. It serves as good communication tools among the stakeholders to discuss application and its future at an abstract level.

**How the architecture vision can be used to reduce the architecture erosion?**
In this study, we have proposed a method named Vision Based Architecture Method (V-BAM) to address architecture erosion problem. In chapter 2 V-BAM method and its implications are discussed in detail. The V-BAM method provides a systematic way to develop and maintain software architecture based on the architecture vision. V-BAM facilitates the production and evolution of high-quality software that meets the needs of its end-users within a predictable schedule and that reduces architecture erosion. V-BAM facilitates the development of software architecture by providing an architecture vision, synchronization of the architecture with the implementation, maintaining architecture design decision and goals of each release separately. In that way software evaluation is performed regularly during each release thus ensuring that any deviation from the architecture is detected in the early phase. That helps to overcome architecture erosion and minimizing the efforts required for building software architecture for the next iteration. In many traditional methods erosion is detected in latter stage that makes it hard to make changes in the system and require extra effort to keep the system alive. V-BAM minimizes the overall efforts that are caused if architecture erosion and drift is not detected earlier.
Chapter 5: Conclusion

Conclusion:

In this thesis we have studied the architecture erosion, its related problems and solutions. We have performed a comprehensive literature survey to identify different reasons for the architecture erosion, discuss how it affects the architecture of the evolving software and how existing studies address the problems of the architecture erosion. We have identified different problems and their causes that arise due to lack or outdated documentation.

In order to address architecture erosion problem we have proposed a systematic method (V-BAM) that helps to manage the software architecture and to avoid architecture erosion and its related problems. This method focuses on the documentation of the changes in the system. To make the documentation simple we have split the architecture document into two separate documents. One is the vision that contains the system level architecture goals, objective, design decision etc. and other document contains the release specific software architecture.

We have performed the static validation of V-BAM on an existing real time system. On the base of this validation we have concluded that the iterative nature and vision based approach of V-BAM provides a clear benefit over the traditional method. The V-BAM helps us to keep the architecture up to date and consistent with the implementation thus reducing or avoid the architecture erosion. We have performed this validation on three releases of the system that just provides the glimpse of the advantages over the traditional methods but there are still quite a lot of factors that needs to be explored.

Future Work

At the moment, we have performed limited static validation of V-BAM. The directions for future work primarily include a more thorough static validation in more than two products involving more people. After that the next logical step is to put this method in real world usage and evaluate the results. This would help to quote quantitative benefits of using this model and return-on-investment for implementing this method. Industrial validation will also help in evaluating the scalability of this method.

Investigate under what condition the architecture vision need to be changed or updated. The architecture vision is not required to be same and it gets evolved with the software and its environment. In addition, identify and categorize the kind of requirements that can affect the architecture vision. Both quality and functional requirements need to be considered in this study.
References


[12] J. S. Poulin, "Evolution of a software architecture for management information systems", Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints ’96) on SIGSOFT ’96 workshops, ACM, 1996


[28]. R. Kazman, H. Peter Inb, Hong-Mei Chenc, "From requirements negotiation to software architecture decisions", Information & Software Technology, 47(8): 511-520, 2005


APPENDIX A

Architecture Vision of the CIS Product

The main goal of vision is to identify major computational and data components of the system under different perspective or context and define clear relationship between them.

Goals and objectives:
1. Aspect oriented techniques will be used to cater different aspects of the application like security, statistics, logs etc.
2. Different design patterns will be used to solve generic problem in a standard way. Like at GUI level, MVC should be used and between the subsystems facade should be used to control dependency and so on.
3. Different components of the software system should be less coupled and more cohesive. Dependency between different components should be controlled and known.
4. System architecture should be developed in the reusable and preferably smaller components.
5. One consistent architecture style should be used throughout the application. In CIS we will use service oriented architecture.
6. Architecture should be able to satisfy and facilitate all the use case scenarios of the system.
7. Use existing and mature third party open source components, libraries and frameworks to reduce development cost.
8. The system should be able to run on any platform (Windows, Linux etc) under minimum constraints.
9. Find more than one way of implementation of each use case and implement the best one with minimum effect on overall architecture of the system.
10. Software architecture should allow different supplier to supply albums and tracks using different format, i.e. system should support for different format.
11. Different components of the system should run in distributed environment.
12. Different component of the system should be able to run as plug and play, where each component is easy to replace with other.
13. Dependences between different components should be defined through the configuration files. Dependency logic should not be embedded in the components. CIS architecture and design should follow the idea of Dependency Injection.
14. Every transaction or activity of the system should be logged.
15. In long term, system should be able to ingest games and other application content in parallel to music contents.
16. There should be easy to use for non technical users.
17. It should be possible to assign roles and rights to the user on every part of the system.

Quality vision:
1. System should be highly scalable; at least 10,000 should be able to access the system at a time without performance hit.
2. The architecture should be structured in a way to achieve high performance.
3. The overall system should be very fast. The album or track should be ingested and published within minimum time period.
4. The down time of the system should not be more than 1 hour in 1 month.
5. The usability of the system should be high; the system should have very rich GUI for each component of the system.
6. Application usage and communication should be highly secured. Different security measures should be used like roles and right, secure certificate etc.
7. Different components should interact using secure protocol and should use different encryption techniques to encrypt and decrypt data.

**Design decision or design/architecture style:**
- Service oriented architecture should be used for CIS, where different component will operate under different services.
- At different level the architecture and design of the system should be developed by keeping change factor in the mind.

**Constraints:**
- Mainly Java and its related techniques will be use for the development of the application.
- Other open source products that are configurable with it could also be considered.

**Frameworks and principles:**
Following frameworks should be use for the development of CIS
- Spring
- Hibernate
- Stripes (MVC)

The system will use following third party libraries for the ease of development.
- Common Apache
APPENDIX B

V-BAM Implementation
In this section we execute the V-BAM method on the CIS product and analyze the implications and advantages of the V-BAM method. Just for the sake of clarity we have combined the implementation of the V-BAM method for all releases. Although in practice the V-BAM method will be implemented separately on each release.

Step1. Identify architecture requirements
In the CIS product the architecture level requirements were not explicitly documented. We have received the requirement specification, code, release notes and change logs for each release from the company. We have used these documents to extract the architecture level requirements of the CIS product for each release.

Architectural requirements – Release 0.5

Functional:
1. The CIS product should be able to import musical deliveries in different formats from different known and unknown suppliers.
2. The system should check the validity and consistency of incoming musical deliveries.
3. In case data is invalid or inconsistent, the system should inform the user and ask about his decision. Once user take any decision about an issue, system should remember it and implement it next time if it faces similar kind of issue.
4. The system should be able to transcode binary files of the musical deliveries into different predefined formats.
5. System should publish albums (or track etc) to different publishers according to the predefined business rules.
6. System should be able to export albums (or track etc) to different media delivery platforms.
7. It should possible to ingest, publish and export album without CIS user interaction.

Quality:
8. For each track transcoding should be performed within 5 min.
9. Each issue should be resolved with in 5 sec after decision is taken by the user.

Architectural requirements – Release 1.2

Functional:
1. The CIS product should have backward compatibility and should satisfy architecture level requirements of the previous releases until explicitly specified to change.
2. It should be possible to configure Imports for different suppliers.
3. Transcoded files should be stored in a dedicated cache.
4. It should be possible to manually clear the contents from the transcoding cache.
5. It should be possible to perform transcoding on the multiple machines.
6. It should be possible to integrate a new command-line interface codec for transcoding.
7. It should not be possible to ingest, publish and export album without CIS user permission.
8. It should be possible for the user to view the published data.
9. The system should maintain the statistic information about the transcoding.

Quality:
10. On average each track should be imported within 10 sec.
11. For each track transcoding should be performed within 2 min.
12. Transcoding should be performed in the secure environment.

Architectural requirements – Release 3.2

Functional:
• The CIS product should have backward compatibility and should satisfy architecture level requirements of previous the releases until explicitly specified to change.
• The system should be able to schedule the publishing of different tracks and albums under the specified business rules.
• The system should maintain the statistic information about the incoming deliveries.
• The publish component should run as a separate service.
• It should be possible to schedule the publishing of album (or track) from the GUI under some specified rules.
• The publisher should be able to purchase individual tracks and/or albums, as well as track and/or album under different offers.
• The publishing component should be configurable.
• The export component should be configurable.
• The export component should be able to schedule exports using GUI under some special rules.
• The system should have ability to cache exports for later use.
• The system should be able to export album or track to different media delivery platforms.
• The system should receive feedback/response from different media delivery platforms against each export and store these responses in the database.

Quality:
• Each track should be imported within 3 sec on average.
• For each track on average transcoding should be performed within 2 min.
• Each track should be published within 5 sec.
• System should use secure protocol to transfer ‘export packages’.

Step2. Develop architecture vision
We have developed the architecture vision of the CIS product by considering the company possible strategies, current and future requirements. Here we have tried to consider the ideal representation of the application and took some assumptions about the future of the CIS product. The architecture vision for CIS product is listed in the appendix A.

Step3. Develop release specific architecture design document
We have extracted the architecture requirements for all three releases from the existing system. Different sub-system, components and interaction among them, are
introduced into the system based on these requirements. The transformation of the requirements of each release on the architecture of the system is presented below:

**Release 0.5**
At conceptual level we have decomposed the CIS product in three sub-systems. The decomposition of the system is performed on the basis of the requirements for the release 0.5 (defined in the step 1) and the architecture vision. As defined in the architecture vision we have applied different design patterns and architectural styles to reduce the dependencies among different sub-systems. At sub-system level we have used Façade design pattern to provide the single interface for the interaction among different sub-system. These sub-systems are further composed of several components, detailed description of these three sub-systems and components are presented below:

**BatchServer:**
The BatchServer mainly address the requirements 1, 2, 3, 5, 6 and 7 as defined in step1, release 0.5. To address these architectural level requirements the BatchServer is further divided into different subcomponent. The architecture requirements and their related subcomponent are presented in table 4.1:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>1</td>
</tr>
<tr>
<td>Validation</td>
<td>2, 3</td>
</tr>
<tr>
<td>Publish</td>
<td>5, 7</td>
</tr>
<tr>
<td>Export</td>
<td>6, 7</td>
</tr>
<tr>
<td>Storage</td>
<td>4</td>
</tr>
<tr>
<td>Drools</td>
<td>2, 3</td>
</tr>
</tbody>
</table>

Table 5.1: BatchServer components and related architecture requirements

**TranscodingServer:**
The TranscodingServer mainly addresses the requirements 4 and 8 as defined in step1, release 0.5. The requirements for the transcoding components are expected to be change in the future so we have decided to keep it independent from the beginning. To address these architectural level requirements the TranscodingServer is further divided into different subcomponent. Transcoding server uses separate database to store the transcoding specific data. The architecture requirements and their related subcomponent are presented in table 4.2:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TranscodingService</td>
<td>4, 8</td>
</tr>
<tr>
<td>TranscodingEngine</td>
<td>4, 8</td>
</tr>
<tr>
<td>Scheduler</td>
<td>4, 8</td>
</tr>
<tr>
<td>Transcoder</td>
<td>4, 8</td>
</tr>
</tbody>
</table>

Table 5.2: TranscodingServer components and related architecture requirements

**GUIServer:**
The GUIServer mainly addresses the requirements 3 as defined in step1, release 0.5. The architecture requirements and their related subcomponent are presented in table 4.3:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 0.5</th>
</tr>
</thead>
</table>

Table 5.3: GUIServer components and related architecture requirements
Release 1.2
The decomposition of the system is performed on the basis of the architecture requirements for the release 1.2 (defined in the step 1) and the architecture vision. In this release architectural requirement does not have major impact on the overall architecture and design of the system. Most of the architecture level requirements in this release are related to the TranscodingServer. We have added/changed or replace some components based on new architectural requirements but overall picture remains the same.

**BatchServer:**
The BatchServer mainly address the requirement 2 and 9 as defined in the step1, release 1.2. To address these architectural level requirements ImportConfig component is added in the BatchServer. In addition to that, Drools component that previously uses the third party component was replaced by the Resolvers component. The Resolvers component provides similar functionality but is not longer dependent on the third party component. The use of third party components in the Drools leads to the performance hit as they consumed lot of memory resources. The architecture requirements and their related subcomponent are presented in table 4.4:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Validation</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Publish</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Export</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Storage</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Resolvers</td>
<td>9</td>
</tr>
<tr>
<td>ImportConfig</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.4: BatchServer components and related architecture requirements

**TranscodingServer:**
The TranscodingServer mainly addresses the requirements 1, 2, 3, 4, 5, 6, 10 and 11 as defined in step1, release 1.2. In the TranscodingServer there were number of different components that are introduced into the sub-system and replaced. Since we have developed the TranscodingServer as separate sub-component it was easy to make changes. In response to requirement number 5, the Transcoder component is replaced by three new components that are TranscoderManager, MasterTranscoder, SlaveTranscoder, Moreover Verifier component is introduced to verify the transcoding performed by TranscoderManager. TranscoderManager component made it possible to achieve the quality requirements 10, as transcoding is now performed on several different machines at the same time. The architecture requirements and their related subcomponent are presented in table 4.5.
Components Requirements – Release 0.5

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TranscodingService</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>TranscodingEngine</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Scheduler</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>TranscoderManager</td>
<td>5, 6 and 11</td>
</tr>
<tr>
<td>MasterTranscoder</td>
<td>5 and 11</td>
</tr>
<tr>
<td>SlaveTranscoder</td>
<td>5 and 11</td>
</tr>
<tr>
<td>Cache</td>
<td>3 and 4</td>
</tr>
<tr>
<td>Security</td>
<td>12</td>
</tr>
<tr>
<td>Statistic</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.5: TranscodingServer components and related architecture requirements

GUIServer:
The GUI Server mainly addresses the requirements 7 and 8 as defined in step 1, release 3.2. The architecture requirements and their related subcomponent are presented in table 4.6:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>GUI Helper</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Security</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Validation</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>DBManager</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>IssueEngine</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Validation</td>
<td>7</td>
</tr>
<tr>
<td>IssueResolver</td>
<td>7</td>
</tr>
<tr>
<td>Publish</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.6: GUI Server components and related architecture requirements

Release 3.2
The decomposition of the system is performed on the basis of the architecture requirements for the release 3.2 (defined in step 1) and the architecture vision. This release had relatively major impact on the architecture of the system. At the conceptual level BatchServer sub-system is split into four different sub-systems that are ImportServer, PublishServer, ExportServer and Core. There are several reasons to split the BatchServer sub-system. As a result of new requirements, the complexity of BatchServer was increasing due to the increased coupling between the components. Service oriented (SOA) architecture is used to reduce the coupling between the sub-systems and make them easy to manage in the future. Moreover the introduction of SOA takes the architecture closer to the vision of the system.

ImportServer
The ImportServer addresses the requirements number 2, 3 and 13 as defined in step 1, release 3.2. ImportServer contains different sub components that are ImportService, ImportEngine, ImportConfiguration, FileManager, Clustering, Statistic, RuleEngine and Parser.

PublishServer:
The PublishServer addresses the requirements number 4, 5, 6, 7 and 15 as defined in step1, release 3.2. The architecture requirements and their related subcomponent are presented in table 4.7:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PublishService</td>
<td>4, 5, 6, 7 and 15</td>
</tr>
<tr>
<td>PublishConfig</td>
<td>7</td>
</tr>
<tr>
<td>PublishEngine</td>
<td>15</td>
</tr>
<tr>
<td>PublishScheduler</td>
<td>5</td>
</tr>
<tr>
<td>RuleEngine</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Publish</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.7: PublishServer components and related architecture requirements

ExportServer:
The ExportServer addresses the requirements number 8, 9, 10, 11, 12 and 16 as defined in step1, release 3.2. The architecture requirements and their related subcomponent are presented in table 4.8:

<table>
<thead>
<tr>
<th>Components</th>
<th>Requirements – Release 3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExportService</td>
<td>8, 9, 10, 11, 12 and 16</td>
</tr>
<tr>
<td>ExportConfig</td>
<td>8</td>
</tr>
<tr>
<td>ExportScheduler</td>
<td>9</td>
</tr>
<tr>
<td>ExportEngine</td>
<td>8, 9, 10, 11, 12 and 16</td>
</tr>
<tr>
<td>ExportCashe</td>
<td>10</td>
</tr>
<tr>
<td>ExportGenerator</td>
<td>11</td>
</tr>
<tr>
<td>Security</td>
<td>16</td>
</tr>
<tr>
<td>DownloadManager</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>FileManager</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>Exporter</td>
<td>Requirements in Release 0.5</td>
</tr>
<tr>
<td>ExportResultHandler</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.8: ExportServer components and related architecture requirements

GUIServer:
The GUIServer addresses the requirements number 4 and 5 as defined in step1, release 3.2. One additional component Export is added in the GUIServer.

**Step4. Develop release specific architecture**
This section presents the architecture of the system that is developed on the basis of architecture vision and architecture design document. For each release the conceptual level architecture diagrams and its rational are given below.

**Release 0.5**
*Architecture Diagram*
Rational:
At conceptual level, the architecture of the system is divided into three sub-systems that are BatchServer, TranscodingServer, and WebServer. The details for each sub-system are as follow.

The basic purpose of the BatchServer is to parse the incoming delivery data and validate it against the existing validated data. If there are inconsistencies in the incoming data it will mark the data as invalid. After performing the validation on the
data, it ingests valid album and tacks into the system. In addition BatchServer is also responsible to publish and export albums/tracks under different business rules. The Import component imports musical deliveries from the outside world into the system. Import component parses the xml files of each delivery, and it contains different parsers for each supplier to parse supplier specific metadata xml files. After fetching data from xml files, Import component transform this data into object form and transfer it to Clustering component. The Clustering component develops the clusters of the incoming data by applying different clustering rules like artist name, country of origin, album or track name etc. Once clusters are created, these clusters are stored in the database and the control is transferred to the Validation component. The Validation component apply different validation rules and pre-existing decisions on the data (called dirty data), and make it clean (consistent and correct data). During the validation process if Validation component faces any issue (like unknown artist name, split artist, unknown country of origin etc) that it could not solve, it passes those issues to IssueEngine component. The IssueEngine takes issues from the Validation component and create related entries into databases for the user to solve them. The Publish component is used to publish albums/tracks to different publisher by applying different publishing rules like in any specific country only authorized publisher should be allowed to publish specific album/track and so on. The Export component takes the published albums and exports it to the different specified media delivery platform (MDP).

TranscodingServer sub-system is used to transcode binary files of musical deliveries into the different formats (mp3, avi, wma etc). This sub-system runs on separate independent machine (server). The TranscodingService component of this sub-system takes deliveries into the system and act as starting point of the transcoding flow. The TranscodingEngine component takes deliveries from TranscodingService and acts as a controller of transcoding system. The Transcoder component is used to transcode deliveries in the different known formats.

The GUIServer sub-system provides a web interface for the user to interact with the system. The user can perform different tasks by using GUIServer. It helps user to view and solve issues created by system etc. The GUIHelper component is used to get data from the backend and to render it on the GUI. The ClientEngine component acts as a controller for GUIServer. The IssueEngine component is used to get issues from database and apply user decisions to solve issues.

**Release 1.2**

*Architecture diagram:*
Rational:
In this release, system still contains three sub-systems although some new components are added to these sub-systems to cater new/changed functional requirements.

In the BatchServer sub-system new component BatchConfig is introduced that is responsible to handle different type of configuration of BatchServer like on/of functionality of system against different supplier etc. The major change in this sub-system is the replacement of existing Drools component with new Resolver component. The Resolver component is used to clean the data by automatically applying pre-existing decisions and new decision taken by the user.

In GUI Server, new Publish component is added to provide publishing interface to the user. Now user is able to write or change new or existing publishing rules/rights dynamically. Publishing component also provide many new functionalities like interface to register, publisher management, rights management of the publishers etc.

The major change is in the TranscodingServer sub-system. New Master, Slave architecture is introduced to perform the transcoding. Now transcoding can be performed in a distributed environment on different machines. In transcodingServer two new components Cache and Statistic are added. The Cache component is used to cache exiting files that helps to satisfy quality requirement of the system. The Statistic component is used to collect different statistic related to transcoding.

Release 3.2
Architecture diagram:
Rational:
In this release there are many architectural level changes. At conceptual level system now contains six sub-systems. Each sub-system is developed as a separate service. Service oriented (SOA) architecture is used to reduce the coupling between the sub-systems and make them easy to manage in the future. ImportServer is more cohesive and only contains the functionally related to the import of the deliveries. The functionalities related to Validation, Publish and Export that was part of ImportServer earlier now works as independent sub-systems. Some new components are added into the system to satisfy new requirements. Here we are going to discuss the components that are new in this sub-system. ImportService component is responsible for handling the services related functionalities. It acts as the starting point of the imports flow. ImportsConfiguration component is introduced to the system to handle all configurations. Static component is responsible to collect all the statistic information related to import and clustering of the deliveries. ImportsEngine component acts as the controller for ImportServer sub-system. It manages the other components and flow of control in the system. RuleEngine component is introduced as the replacement of old validation and Resolver components.

GUIServer does not have many changes at architecture level, although there are a lot of changes at design level in different components like publish, IssueEngine etc. A new component is introduced into the GUIServer with the name Export. Export component provide export interface to the user. Through export interface user can manage and schedule exports and many other task related to exports.

Core sub-system is responsible to manage and control other sub-system of the CIS product. It works like an administrator in the CIS system. This Core sub-system is also responsible to manage many system level configuration of the CIS product. SystemConfiguration component manages most of the configuration of the system. CoreConfiguration component handle configuration of Core sub-system. InitService
component is responsible to start and stop different services in CIS-system. It acts as starting point of the system. The ValidationService is responsible to perform validation of the incoming data and to resolve existing issues in data. It gets request from the GUIserver and ImportServer. The validationEnigne component performs validation on the incoming data from ImportServer. Resolver component is responsible to resolve issues by using existing decision or by getting decision from GUIserver. StorageService component is a service that is initiated by InitService. On the successful ingestion of the delivery, StorageEngine component transfers that delivery/s to transcoding server.

The PublishServer is used to handle the publishing of the deliveries in the CIS product. PublishConfig component handles the configuration of this sub-system. PublishEngine acts as core part of PublishServer sub-system. It also manages and controls other components in PublishServer and communication among them. The RuleEngine component is used to implement different static and dynamic publishing rules on the album or track. PublishingSecheuler component is responsible to schedule publishing of different albums and tracks. Publish component is used to publish album any album or track to different publishers.

ExportServer sub-system is used to export the ‘export package’ on the media delivery platform (MDP). ExportService component is the starting point of this sub-system. It interacts and receives the export request from outside world. ExportConfig component handles the configuration of the ExportServer sub-system. ExportScheduler is responsible to schedule different exports. The scheduling depends on the different business rules, deliveries and publishers. ExportEngine component acts as core component of ExportServer that is used to manage and control different components in ExportServer sub-system. DownloadManager component is used to download different album and tracks from TranscodingServer to ExportService. ExportGenerator component is responsible to package different albums and tracks in one export package. It creates export packages specific to the publishers. ExportCache component is used to cache exports. It helps to increase performance of ExportServer. The Exporter component is responsible to dispatch export packages to different MAP. The Security component is used to implement encryption and decryption on exports. The ExportResultHandler gets results or response from MDP against each export package and store it into the database for later use.

**Step5. Synchronize architecture with vision and architecture design document**

In this step we have analyzed the architecture of the previous release to find any deviation from the architecture that was supposed to be implemented. Once we have identified the deviation we have compared the implemented architecture with the architecture vision. On the basis of these findings we have designed the architecture for the next release in a way that will reduce the gap between the implemented architecture and architecture vision.
GLOSSARY

Meta data:
It explains the meaning of some terminologies used in the description of requirements above.

Musical delivery:
Usually Musical delivery exists in the form of folder in the system. A typical musical delivery contains binaries (songs, videos etc) and xml file/files that contain data about binaries.

Supplier:
The companies that supplies musical deliveries like Universal, Sony BMG, and Orchard etc.

Clean data:
The data that is valid and consistent with rest of the system.

Dirty data:
It represents the data that is not valid and might not be consistent with rest of the system.

Issue:
The problems found in the validity and consistency of data.

Publish:
Apply different rules and principles to allow publisher to purchase the contents.

Export:
The export creates exports for upload to target platform. Each export shall consist of one XML metadata file with valid file references to the related binaries.

Transcoding:
Transform one musical file (song etc) into different formats those are compatible in different platform.