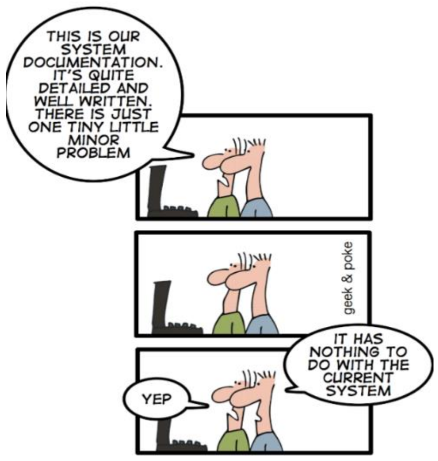

THIS IS OUR SYSTEM DOCUMENTATION. IT'S QUITE DETAILED AND WELL WRITTEN. THERE IS JUST ONE TINY LITTLE MINOR PROBLEM
geek & poke
IT HAS NOTHING TO DO WITH THE CURRENT SYSTEM
YEP

A literature study of architectural erosion and comparison to an industrial case in Danfoss.

Group: Alpha

Thanh Cong Le 20035165 thanhletcl@gmail.com

Department of Computer Science,
University of Aarhus
Aabogade 34, 8200
Århus N, Denmark

Oktober 2017

Abstract

Software architectural erosion is a big recurring problem for every software project. It has caused increasing interest in how to detect, stop and reverse the architectural erosion. However, up until now, there have not been many attempts to obtain an extensive overview of the work in the field. In this report, we are using systematic review technique to classify and report the reasons for architectural erosion and techniques for detecting architectural erosion. We will also present a survey of approaches that have been proposed over the years for avoiding architectural erosion. Danfoss AK2 system serves as a case study for the similarities of why the architectural erosion happens.

Contents

Disclaimer

1. Motivation

Danfoss is a global leader focused on energy-efficient solutions that save energy and costs, and reduce carbon emissions. The company's wide range of products and services are used in areas such as cooling food, air conditioning, heating buildings, controlling electric motors and powering mobile machinery. Danfoss Cooling division covers a wide range of applications within refrigeration segments.

[Confidential material removed]

By nature, over time it is expected that the software will evolve, because new requirements will emerge. The wish is to know and understand the root causes to be able to keep the architecture under control even though it is evolved.

It would be very interesting to identify the challenges the company needs to face to keep the architecture and the code maintainable. During this literature study and by using tools and principles from my Master IT courses I would like to study and investigate to gain a better understanding of the reasons for why the architecture erosion occur and based on that knowledge, find a way to detect and stop software architecture erosion.

I believe that in the next five years, more software systems will be introduced and software will be a major part of our daily environment, therefor it is important that we can gain more knowledge about the architectural erosion to keep the software systems in a good shape.

Another motivation is by using one of our current software architecture as a case study to gain the knowledge for how to keep the software architecture under control within a big company as Danfoss using the existing technologies.

I am using the Danfoss AK2 system, which is described in section 2 as a case study to investigate if the reasons for architectural erosion are also found in the AK2 system.

1.1 Readers guide

The remainder of this report is structured as follows:

- Section 2 describes the AK2 system, which is used as our case study in section 6.
- Section 3 describes the research method used in this report.
- Section 4 presents the detailed description of our systematic literature review.
- Section 5 presents the background definitions and theories about the architecture and architectural erosion.
- Section 6 presents the first part results of the review with detailed description of architectural erosion root cause's classification. ***Figure 11*** visualizes the conclusion of report's hypotheses H1. A comparison to the AK2 system is also presented here.
- Section 7 presents the second part results of the review in the detecting techniques classification. The conclusion of report's hypotheses H3 is also visualized in *Figure **14***.

- Section 8 presents the list of existing techniques to avoid the architectural erosion.
- Section 9 provides a quick **summary** of the **report's** content.
- Section 10 contains the discussion and conclusions drawn from this study together with the future works.

1.2 Terms

Evolution	The process of developing software initially, then repeatedly updating it for various reasons. [1]
PLM	Product life-cycle management
CRM	Customer Relationship Management
ClearQuest	Configured as a bug tracking system
AK2	AdapCool Product
DNIP	Danfoss communication protocol
O&O	Object Oriented
Agilists	is an enthusiast of Agile techniques for management of software.
PMI	Product Maintenance Innovation
NPD	New Product Development

2. Problem statement

The problem with architecture erosion is that it will accumulate over time until any change in the architecture is a risk for the whole system. By identifying the reasons for the architecture erosion we can do some pro-active actions to prevent or slow down the erosion.

It is also a need to detect the architecture erosion to be able to take steps to actively stop architecture erosion.

By collecting and studying papers and books together with the experience from my own work, I want to:

- (H1) Report and classify the reasons for architectural erosion as reported in the research literature and current Software Architecture books.
- (H2) Compare and identify the similar architectural erosion reasons in the AK2 system.
- (H3) Report and classify the techniques to detect architecture erosion.
- (H4) Perform a survey of the available approaches to avoid architectural erosion

3. Method

As this is a literature study and a lot of information needs to be obtained and processed, we will use the procedure from Procedures for Performing Systematic Reviews [2] as a guideline and inspiration on how to perform systematic review the papers.

Searching and selecting the books and papers are filtered and focusing around the Software Architecture Erosion topics.

Besides classifying the reason for architectural erosion and the detecting techniques from the selected materials, we will also investigate if the architectural erosion in our case study has the same similarity.

Based on the gained knowledge, a list of the techniques for avoiding architectural erosion will be generated.

3.1 Delimitations

This master thesis is subject to the following delimitations:

- We will not cover the tools' availabilities aspect for detecting and avoiding architectural erosion.
- We will not cover the architecture recovery technique.

3.2 Validity threats

The main threats to validity in this systematic review are bias in our selection of the literature to be included, and data extraction. To be able to identify relevant studies and ensure that the process of selection was unbiased, the review process is defined in the beginning to define research questions, inclusion and exclusion criteria.

By limiting ourselves to only one case study, we risk that conclusions may not be applicable to other domains and companies. The corporate cultures, the domain Danfoss is operating in and inside knowledge by self-working with the study case, can affect our conclusions.

4. Review process

We undertake the literature study by defining the review process inspired by Procedures for Performing Systematic Reviews [2]. The review process includes four phases and several steps:

- **Review process**
  (a) Define review process.

- **Define search criteria**
  (b) Define study question.
  (c) Define search terms.
  (d) Define inclusion and exclusion criteria.
  (e) Define quality assessment criteria.

- **Literature search**
  (f) Perform searching process.

- **Data**
  (g) Reading and data extraction.

The whole process is illustrated in ***Figure 1 Systematic literature review steps***. In the following sections, we will provide more details about the individual steps.

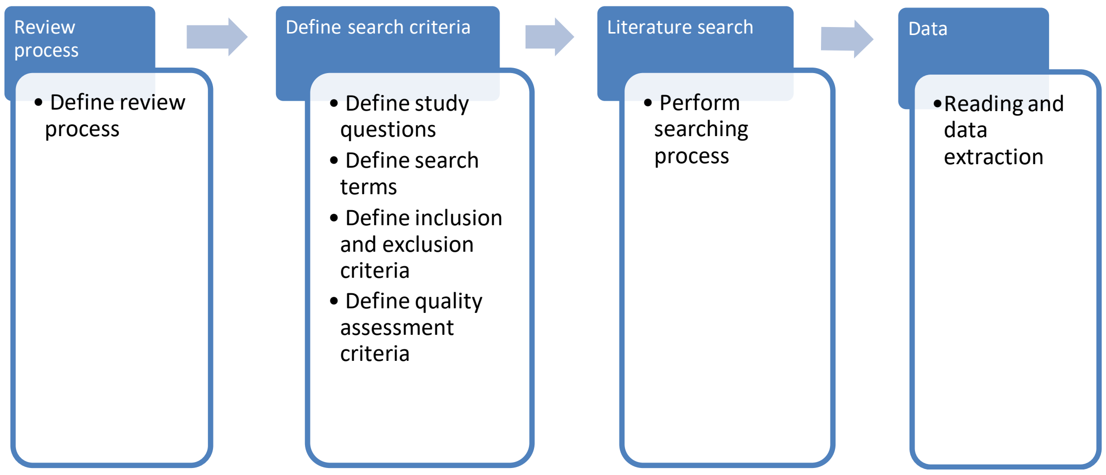

Review process
• Define review process
Define search criteria
• Define study questions
• Define search terms
• Define inclusion and exclusion criteria
• Define quality assessment criteria
Literature search
• Perform searching process
Data
•Reading and data extraction

Figure 1 Systematic literature review steps

4.1 Review process

4.1.1 Define review process

This step is to define and specify the whole process and the goals for each step. Study question are defined as one of the first step as it is a basic of our study. All the steps are described in the next following sections.

4.2 Define search criteria

4.2.1 Define study questions

The main objective of this study is to report and classify the reasons for architectural erosion as reported in the research literature and current Software Architecture books. Secondary is to report and classify techniques to detect architectural erosion. Third is to list all existing techniques /approaches to avoid architectural erosion. Based on those objectives we formulate following study questions:

1. Why does Software Architectural erosion happen?
2. How to detect Architectural erosion?
3. How to avoid Architectural erosion?

4.2.2 Define search terms

To ensure a consistent search result, we define the search terms in the beginning. The initial search is performed on www.google.com, which results into many hits, after following a couple of links we can conclude that most papers primary is placed compactly in only a few electronic databases. It is clear, that those scientific electronic databases can provide the most important and with highest impact full-text journals, papers and conference proceedings, covering the fields of software quality, software architecture and software engineering in general. Therefor we have decided to perform the search only on those scientific electronic databases with complementary search using www.google.com.

Following search terms are used to find relevant materials:

- Search 1: Software architecture AND erosion
- Search 2: Stop AND Software architecture AND erosion
- Search 3: Architectural consistency
- Search 4: Architecture degradation
- Search 5: Software AND erosion

After reading a few papers, we have realized that there are different terms, which are used for architectural erosion. We decided to add those search terms:

- Search 6: Architectural decay
- Search 7: Design AND erosion
- Search 8: Design AND decay

More details about searching process is presented in section ***4.3.1 Performing searching* process**

Following electronic databases are chosen:

- Google Scholar: scholar.google.com
- ACM Digital Library: http://portal.acm.org
- Compendex http://www.engineeringvillage.com

- IEEE Xplore: http://www.ieee.org/web/publications/xplore/

4.2.3 Define inclusion and exclusion criteria

The goal of setting up criteria is to find all relevant literature in our study and reduce the amount of the literature we need to screen. We don't set up the lower boundary for the year of publication, because we want to include all the relevant literature that is stored in the database over years. We think that in the matter of architectural erosion there are not any step changes, which overthrow any old researches. Since English is our main language we only consider books, journals, conferences, report and workshops in English.

We exclude all the papers, which are not explicitly related to architectural erosion. We also exclude prefaces, editorials, summaries of tutorials and duplicated literature, which exist in different versions. For the duplicated literature, we only include the most complete and up-to-date version and exclude the others. All the literature must satisfy all inclusion criteria and not satisfy any exclusion criteria. The defined inclusion and exclusion criteria are shown in ***Table 1 Inclusion and exclusion criteria***

Inclusion criteria	English book, journals, report and conference answers to the study questions.
	Papers, that focus on architecture erosion
Exclusion criteria	Prefaces, editorials, summaries of tutorials
	Papers are not in English
	Duplicate papers.

Table 1 Inclusion and exclusion criteria

4.2.4 Define quality Criteria

To ascertain our confidence in the quality of the found literature, we use the following quality criteria for appraising the selected book, journals, report and conference:

a. The paper is cited in minimum 2 other papers.
b. The paper has a description of the context.
c. The studies methods were well defined and deliberate.
d. The study environment and contexts are clearly stated.
e. The observations/results support the conclusions.

All the included papers shall meet all above criteria. The citation rates of the included papers numbers are obtained and verified using information from Google Scholar and IEEE *Xplore* Digital Library.

4.3 Literature search

4.3.1 Performing searching process

We divide the search process in two phases: **initial searching** and **reference based searching**. The reason for performing the search process in two phases at two points in time is to ensure we didn't miss anything and we have discovered the latest result of literature.

The searching process contains a few steps, which we execute during the searching process.

Figure 2 Initial searching shows the search process and the number of found papers on each step in the first phase. In the first step, we perform the search on the selected electronic databases using the search terms. The first search in the selected electronic databases returned with around 4700 papers. These papers are filtered based on title, description and abstracts, and then they are checked against the inclusion and exclusion criteria. Duplicated and irrelevant papers are removed and this results in around 300 remaining papers. After a quick text screening of the papers, we reduce the amount of papers to 155 papers, which we entered into the tool EndNote, which was also used in the subsequent steps for reference storage and sorting. Those are again reduced further to 70 by full-text screening to ensure that the contents are really related to the search questions.

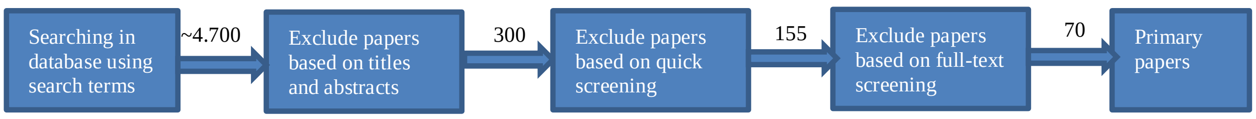

Searching in database using search terms
~4.700
Exclude papers based on titles and abstracts
300
Exclude papers based on quick screening
155
Exclude papers based on full-text screening
70
Primary papers

Figure 2 Initial searching

During the full-text screening, we noted all references in the paper, which related to the search questions and satisfied our inclusion and exclusion criteria. It results in 204 papers, which we again perform the searching process steps on. The number of found papers on each step is illustrated in ***Figure 3. Reference based searching.***

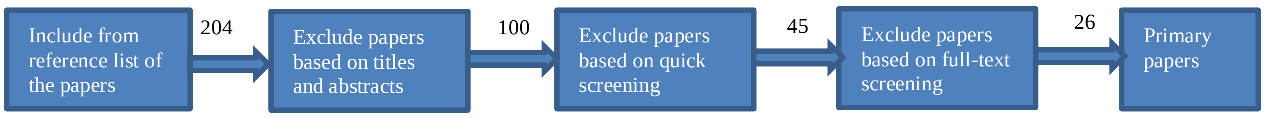

Include from reference list of the papers
204
Exclude papers based on titles and abstracts
100
Exclude papers based on quick screening
45
Exclude papers based on full-text screening
26
Primary papers

Figure 3. Reference based searching

Again, during the full-text screening, we discover new terms for architecture erosion, which forces us to perform additional searching based on the new terms. Those new searches are performed following the entire search process. We conclude that there are too many duplicates, but we found 12 more papers. This resulted in a total of 108 papers in the final list.

4.4 Data

4.4.1 Reading and Data Extraction

The whole list of papers is added to bibliographical management Tool Endnote to manage the bibliographical list. To maintain the relevant list of reference, the same literature list is added to the Excel spreadsheets. The primary literature list is presented in ***Appendix A***. The data extraction and processing is carried out by reading each of the 108 papers thoroughly and extracting relevant data. The data will be presented in section 6 and section 7.

Before presenting the review results, we want to introduce the background definitions and theories about the architecture and architectural erosion in the next section.

5. Software Architecture

5.1 What is Software Architecture?

There are many different definitions of software architecture. There are even websites, which maintains collection of software architecture definitions. [3]

In the book, *Software Architecture in Pratice* Bass, Clements, and Kazman define architecture as follows:

"The software architecture of a program or computing system is the ***structure*** *or* ***structures of the system****, which comprise software* ***elements****, the externally visible* ***properties*** *of those elements, and the* ***relationships*** *among them. Architecture is concerned with the public side of* ***interfaces****;"* [4]

McGover has another definition:

"The software architecture of a system or a collection of systems consists of all the important ***design decisions*** *about the software structures and the* ***interactions*** *between those* ***structures*** *that comprise the systems. The design decisions support a desired* ***set of qualities*** *that the system should support to be successful. The design decisions provide a* ***conceptual basis*** *for system development, support, and maintenance."* [5]

According to ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems, a software architecture:

"is defined by the recommended ***practice*** *as the fundamental organization of a system, embodied in its* ***components****, their* ***relationships*** *to each other and the environment, and the* ***principles*** *governing its design and evolution."* [6]

Kruchten is also focusing on terms: elements, behaviors, decisions and architectural style. His definition is:

"An architecture is the **set of significant decisions** *about the organization of a software system, the selection of* **structural elements** *and their interfaces by which the system is composed, together with their* **behavior** *as specified in the collaborations among those elements, the* **composition** *of these elements into progressively larger subsystems, and the* **architectural style** *that guides this organization -- these elements and their interfaces, their collaborations, and their composition."* [7]

Object Management Group Inc. emphasizes also the system's structure and behavior together with component's interfaces and relationships in their definition of software architecture:

"[Architecture is] the organizational ***structure*** *and associated* ***behavior*** *of a system. An architecture can be recursively decomposed into parts that interact through* ***interfaces****,* ***relationships*** *that connect parts, and constraints*

*for assembling parts. Parts that interact through interfaces include **classes**, **components** and **subsystems**."* [8]

As we have emphasized with the bold words, even though all those definitions are kind of different, but the similarity in all of them is the notion that, the architecture of a system describes its structures, elements, relations and decisions using one or more views. This means the software architecture is knowledge about a software system, a knowledge, which includes everything that leads to the software system's current form. The software architecture can be considered as a bridge between requirements and the implementations.

The ***Figure 4*** illustrates the relationship between business goals and software architecture, it contains the system qualities as performance, modifiability, testability, security and usability [4], which is also known as quality attributes. We will look more into the software architecture views and quality attributes when we go through the definition of well-documented software architecture.

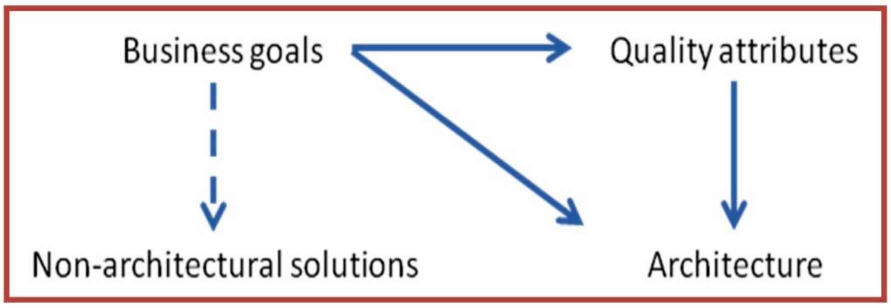


Figure 4 Business goals and architecture [4]

By representing a common abstraction of the system and as a central knowledge about a software system, the software architecture is used by the architects, developers and stakeholder for communicating, reasoning and learning about the system. The use of software architecture spans over the whole software system life cycle, but it is the most important artifact in the first development phase, since the software architect is not only used for early analysis to ensure that a design approach will result into an acceptable system. But also, to capture the earliest design decision, which will have a big impact on remaining development, deployment and maintenance phases.

In this report we follow the definition of architecture as it is used in the ANSI/IEEE Std 1471-2000, the architecture:

*"is defined by the recommended **practice** as the fundamental organization of a system, embodied in its **components**, their **relationships** to each other and the environment, and the **principles** governing its design and evolution."* [6]

5.2 What is good software architecture?

According to Bass et al. [4] there is no bad or good software architecture. Architectures are either more or less fit for some stated purpose and can only be evaluated in the context of specific goals. But Bass et al. defined some rules of thumb that should be followed when designing architecture to avoid a flawed architecture. Those rules of thumb are divided into two categories:

Process rules of thumb:

- The architecture should be the product of a single architect or a small team with an identified leader.
- The architect (or the team) should have the functional requirements and quality attributes (prioritized).
- The architecture should be well documented.
- The architecture should be reviewed with the stakeholders.
- The architecture should be evaluated for quality attributes.
- The architecture should lend to incremental implementation (via the creation of a "skeleton" system).
- The architecture should result in a specific set of resource contention areas. The resolution of which is clearly specified, circulated and maintained.

Structural rules of thumb:

- Well defined modules whose functional responsibilities are allocated on the principles of information hiding (including abstraction of computing infrastructure) separation of concern.
- Each module should have a well-defined interface.
- Hiding changeable aspects allows the development teams to work independently.
- Quality attributes should be achieved using well-known architecture tactics specific to each attribute.
- If architecture depends upon a commercial product, it should be structured such that changing to a different product is inexpensive.
- Creating / Consumption of data should be separated in different modules.
- Parallel-Processing modules: well defined processes or tasks that do not necessarily mirror the module decomposition.
- Task/Process assignment to processor should be changeable.
- The architecture should feature a small number of simple interaction patterns.

In this report we consider well-documented software architecture as the most important rule to achieve a good architecture and to avoid architectural erosion. Addition to that, we also mean that architecture needs to be managed through its whole lifecycle. In the following sections we will present both the definition of well-documented software architecture and the software architecture development lifecycle.

5.2.1 Well-documented software architecture

The code cannot tell the whole story, according to Simon Brown [9] there are several layers of information above the code to provide the full picture of the system. Those layers provide the answers to the questions, which are illustrated in ***Figure 5 Additional layer of information above the code***

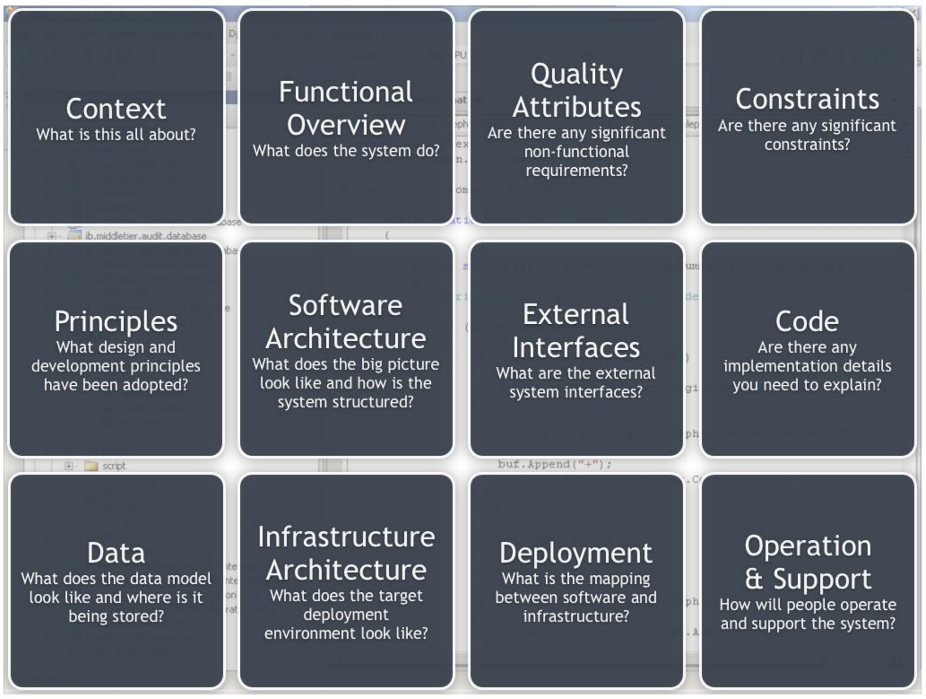

Context
What is this all about?
Functional Overview
What does the system do?
Quality Attributes
Are there any significant non-functional requirements?
Constraints
Are there any significant constraints?
Principles
What design and development principles have been adopted?
Software Architecture
What does the big picture look like and how is the system structured?
External Interfaces
What are the external system interfaces?
Code
Are there any implementation details you need to explain?
Data
What does the data model look like and where is it being stored?
Infrastructure Architecture
What does the target deployment environment look like?
Deployment
What is the mapping between software and infrastructure?
Operation & Support
How will people operate and support the system?

Figure 5 Additional layer of information above the code **[9]**

Documenting architecture is a matter of documenting the relevant views and then documents the information that applies across the views [10]. These views and information is a vital part to communicate and to provide an overview of the whole system. If we look solely at the software architecture document, they must provide an architectural overview of the system together with the significant architectural decisions, which have been made on the system.

Bass et al. mentioned architecture documentation as architectural description [4]. It is the same term the International Organization for Standardization used in ISO/IEC 42010 standard [11]. The ISO/IEC 42010 standard defines a conceptual model for architecture documentation as it illustrates in ***Figure 6 A Conceptual Model of Architecture Description.***

The process of creating an architecture description usually starts with identifying stakeholders of the respective system and their concerns. In a project, there are several types of stakeholders, who have their own views, their own interests and their own concerns. These views address different concerns of multiple stakeholders. Different types of views are described by viewpoints.

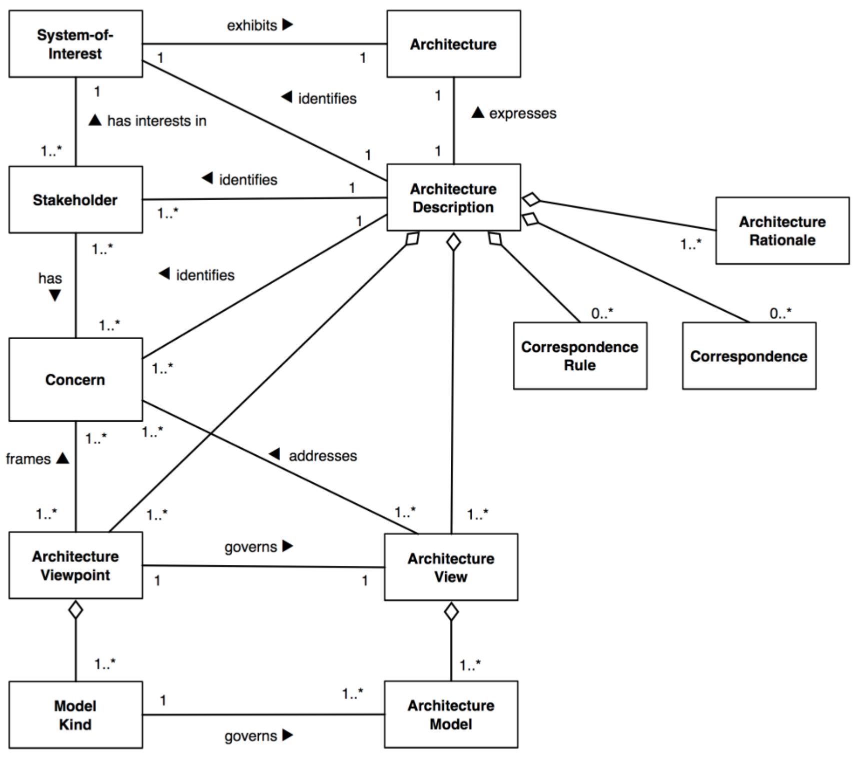


Figure 6 A Conceptual Model of Architecture Description **[10]**

This conceptual model is not only used by Kruchten, who defines the 4 + 1 View Model on Software Architecture [4], but also widely used in different architecture documentation approach like the Siemens Four-Views Model [12], the SEI Views and Beyond Model [12], and 3+1 views [13], which bases on Three viewpoints [10] (see ***Figure 7 Ontology of architectural descriptions***). In the section ***5.2.3 Views,*** we will provide more details about those views.

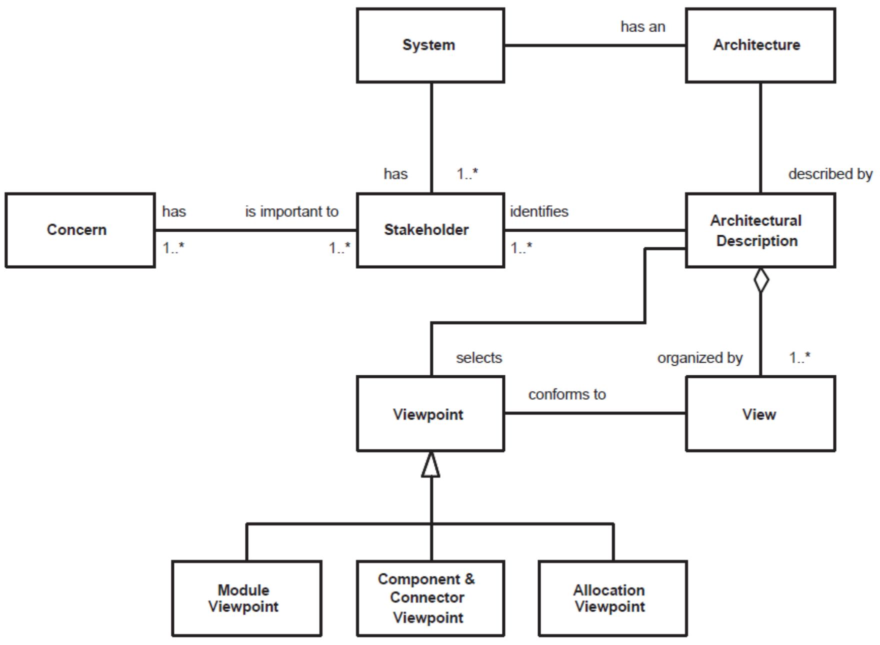


Figure 7 Ontology of architectural descriptions

To make a well-documented architecture we will need to understand the usage of the architecture documentation. In the book *Documenting Software Architectures: Views and Beyond* [10] Clement et al. describe the three usage of archeticure documentation as following:

1. Architecture serves as a means of education.
   The architecture documentation is an education starting point for the new people, who join the project. Therefor the architecture documentation must not only provide a high-level overview of the system, but also a key to find other relevant information such as decisions, detail design document etc.
2. Architecture serves a primary role as a vehicle for communication among stakeholders.
   The architecture documentation will be used communicate with all the stakeholders, for example, negotiating trade-offs or handing off detailed design and implementation tasks.
3. Architecture serves as the basis for system analysis.
   The architecture documentation will be used for early analysis, for example performance analysis and validation of the architecture against the system quality attributes.

In this report, we define well-documented architecture as defined in ISO/IEC/IEEE 42010:2011 [11], which contains:
- Stakeholders and concerns.
- Quality attributes.
- Viewpoints and views.
- Architecture decisions and rationale.

Once again, we will emphasize the importance of documenting the relationship between business requirement, quality attribute and the viewpoints as illustrated in ***Figure 4 Business goals and architecture*** . To get a better understanding of when the documentation phase start, in the following sections we will describe the software architecture development lifecycle and look into views, which are the most important part of architecture documentation. We will conclude the sections with our thoughts on the architecture documentation matter.

5.2.2 Software architecture development lifecycle

The software architecture development lifecycle is composed of a set of phases illustrated in ***Figure 8 Phases of the software architecture development lifecycle***: architectural requirements analysis, architectural design, architectural documentation, and architectural evaluation [14].

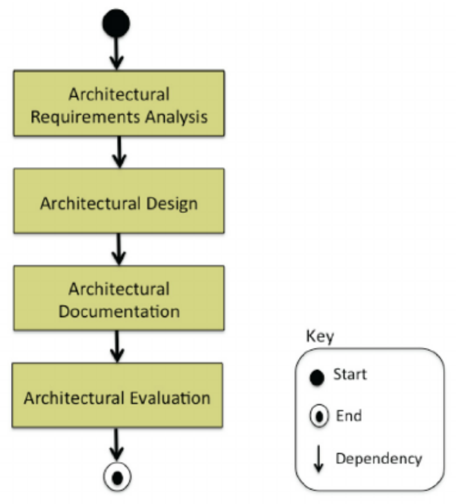


Figure 8 Phases of the software architecture development lifecycle

During Architectural Requirement Analysis, architecturally significant requirements are identified. The phase starts with mapping the business goals into quality scenarios or use cases, which can help to determine the architectural drivers and business goals. The architectural drivers are the architectural requirements, which have the most impact on the architecture. This is done by involving stakeholders in activities such as analyzing, specifying and prioritizing architectural requirements. These requirements are both functional and none-functional requirements. The non-functional requirements are also including both constraints and quality attributes, which are used to drive the design of the architecture.

There are two different method to approach this phase *Quality Attribute Workshop(QAW)* [15] and *Architecture Centric Design Method(ACDM)* [16]. In QAW quality attributes requirements are specified as scenarios, which are textual descriptions of how the system responds. In ACDM, there focus more on architecture design and evaluation.

Once the architectural drivers are identified the architectural design phase can begin. At this time point, the patterns and tactics will be chosen to control of quality attribute response.

After the design phase, the architectural documentation will be started and the software architecture development is finishing with the architectural evaluation. *Architecture Tradeoff Analysis Method(ATAM)* [17] is one of the methods to be used in this phase.

The Architecture development lifecycle illustrated in ***Figure 8 Phases of the software architecture development lifecycle*** is a very simple model. Christensen et al [18] present a more detailed model, where there are evaluation activities, architecture management and architecture interaction following after each of three main steps; architectural analysis, architecture design, and architecture realization.

5.2.3 Views and viewpoints

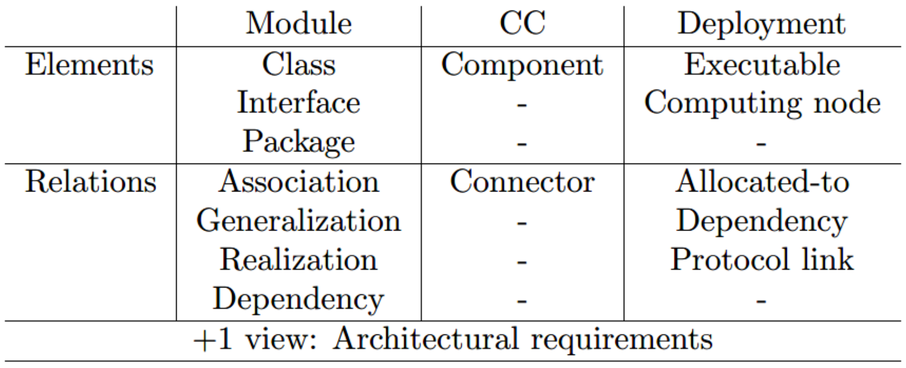
The views are the most important part of architecture documentation. Views and viewpoint concept is based on Kruchten's suggestion in his paper *Architectural Blueprints—The 4+1 View Model of Software Architecture* [19]. The IEEE standard makes this idea generic and introduces the viewpoint concept. IEEE defines view and viewpoint as following:

*A **view** is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholder.* [11]

*A **viewpoint** is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views.* [11]

Because of complexity of the system, a view cannot represent all the details of the whole software architecture, a set of views are required to represent the whole system. Kruchten defined the 4+1 view model, which contains four standard views, namely, Logical, Process, Physical, and Development. The fifth view corresponds to the Use Case view around which the other views revolve [19].

Christensen et al. presented a similar approach with 3+1 view model [13]. The overview of the three viewpoints: Module, Component and Connector and Deployment together with their associated elements and relations are shown in ***Figure 9 3+1 View Model overview.*** Same as the 4+1 view model, the 3+1 view model also uses UML as the syntax for documenting the architecture.



	Module	CC	Deployment
Elements	Class	Component	Executable
	Interface	-	Computing node
	Package	-	-
Relations	Association	Connector	Allocated-to
	Generalization	-	Dependency
	Realization	-	Protocol link
	Dependency	-	-
+1 view: Architectural requirements			

Figure 9 3+1 View Model overview [13]

The Module viewpoint is a static view of the system, where it visualizes the class, package and interface of the implementation.

The Component and Connector viewpoint is a dynamic view of the system. It visualizes the components, which are units of functionality and connectors, which are representations of communication paths between components.

The Deployment viewpoint is concerned about how the software elements are mapped to the hardware.

In this report, we are following the 3+1 view model.

5.2.4 Reflection

[Confidential material removed]

A well-documented architecture is the one, which provides enough documentation for stakeholders to understand the architecture. The architecture is only useful if it is understood by its stakeholders. Therefor the documentation must contain sufficient details for early analysis, but at the same time keep a high abstract level for the overview of the architecture. Well-documented architecture is an important asset, helping to minimize the cost of software maintenance and managing evolving software systems [20].

But having a well-documented architecture in the beginning is not enough for the whole architecture life cycle; the documentation is worthless if it is not up to date. According to Parnas, by keeping documentation up to date, it can improve the software [21]. Because during a detailed and systematic documentation of the code, the developers will often be able to reveal bugs, duplications or almost alike functions, and sometimes another ways to improve performance. But documenting design and architecture and keeping it up to date is not an easy and trivial task. Developers maintaining a product are often not involved in the initial development phases. In these cases, the lack of up to date documentation complicates product maintenance, extensibility and also forcing developers to make assumption, which can result into an architectural smell.

Besides requiring a big effort to keep the documentation up to date, there is also another reason for the developers to skip documenting, often because of time pressure in the projects. If a release-date or deadline comes closer, workarounds are implemented to get the product running, no one wants to document a workaround. After the software release, nobody will remove these violations because refactoring, testing and documenting can result in significant effort. This phenomenon is **Technical debt**, which not many managers and organization recognized or accepted.

It is important to keep in mind that the use of well documentation will not solve all communication problems. Even though the documents are great, they still require a good communicator to present them to the stakeholder in an understandable way.

> ***The primary problem with documentation is the difference between context and content. Documentation can provide content, but understanding the context requires domain expertise.***

"Agile Project Management, Creating Innovative Products", Jim Highsmith [22]

Studies of the cost for software maintaining have shown that over 50% of the system maintainer's time is spent in the process of understanding the code that they are to maintain [23]. Therefor it is an importance to keep the document up to date to ease their task, thus reducing both effort and cost.

5.3 Software Architecture Erosion

5.3.1 Definition

No architecture is a onetime shot. The system will be requested to add more features or improve different quality attributes like performance or availability. The architecture needs to change to adapt to those new requirements. This is called **evolution** and this where the erosion happens.

Perry and Wolf [24] distinguish between ***Architectural drift*** and ***Architectural erosion. Architectural erosion***, according to Perry and Wolf, is the process of introducing changes into a system architecture design that violates the rules of the system's intended architecture.

Architectural drift, on the other hand is the introduction of design decisions into a system's architecture that were not included in the intended architecture, they do not violate any of the previously-made design decisions.

Even though the software architectural erosion are known issue in the software development field, but they appear in different books, reports and papers with different terms as software architecture erosion [25], architectural decay [26], architectural degeneration [27], software erosion [28], design erosion [29] and design decay [30].

Taylor at al.'s definition of architecture erosion focusses more on the ***prescriptive architecture*** and ***descriptive architecture***, where the prescriptive architecture captures the design decisions made prior to the system's construction, this architecture is known as ***as-intended*** architecture. While the descriptive architecture is more about describing how the system has been built, it is known as ***as-implemented*** architecture. Taylor at al. defines architecture erosion as "*Architectural erosion is the introduction of architectural design decisions into a system's descriptive architecture that violates its prescriptive architecture*" [31].

In ***Figure 10. Loss of architectural knowledge***, at the left illustrates the ideal situation, when all the developers possess accurate knowledge about the system architecture and both the prescriptive and descriptive architecture is the same. But at the right side of the figure illustrates the situation, where only a part of the intended architecture is documented and only a part of the descriptive architecture is implemented compared to the prescriptive architecture. Furthermore, it also illustrates the loss of architectural knowledge as a reason for that; different developers understand the system architecture differently and none of them having an accurate view of the prescriptive architecture, which cause the architectural erosion. We will cover more about the reasons for architectural erosion in ***section 6***.

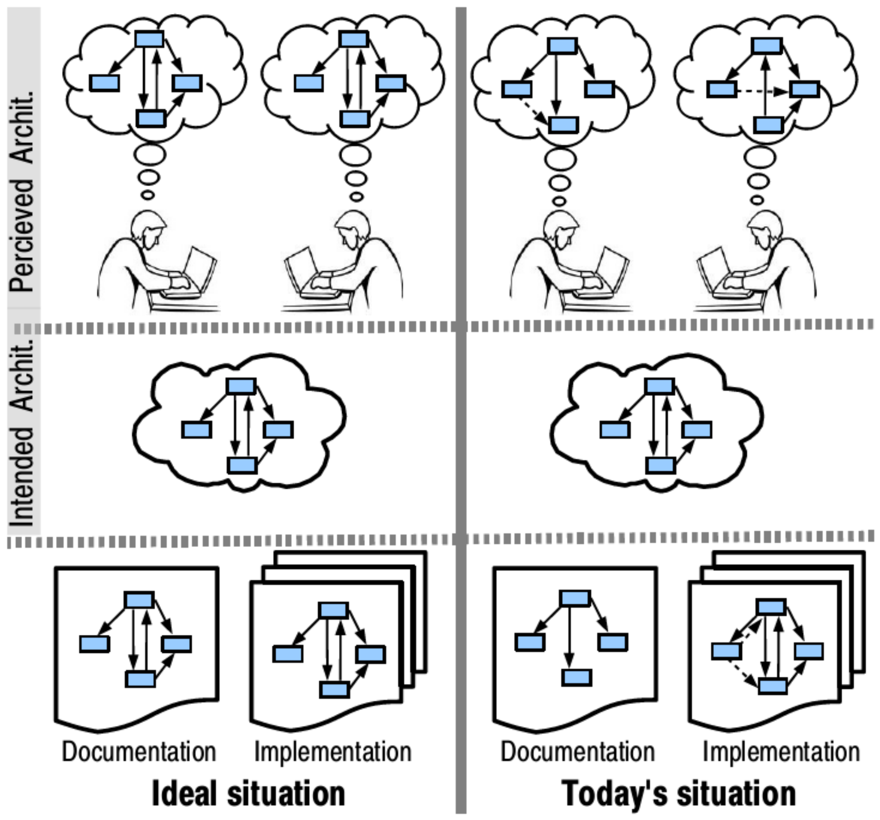


Figure 10. Loss of architectural knowledge [32]

There are two types of the architectural violations: divergences and absences. Divergences are when the relations and dependencies are not prescribed in the prescriptive architecture, but are present in the descriptive architecture. Absences are when the relations and dependencies are prescribed in the prescriptive architecture, but are not present in the descriptive architecture.

Dalgarno [28] further defines the common type of software architecture erosion:

- ***Architectural Rule violations*** the design rules are not followed e.g. where strict layering between subsystems is bypassed.
- ***Cyclic dependencies*** – is one of the worst type of erosion, because it easily sneaks into the design. For example, A calls B, B calls C, C calls D, and D calls A. This type of dependency can be valid but when it's unintended can lead to very complex, opaque code that is hard to understand and hard to test in isolation.
- ***Dead code*** – code that once supported part of the software, is now no longer used, but is still messing the code base contributes towards architectural erosion.
- ***Code clones*** – also known as ***'Copy & Paste'*** Codes, identical or near-identical code fragments scattered across the system. A bug fix or change in one clone instance is likely to have to be propagated to the other clone instances.
- ***Metric outliers*** e.g. very deep class hierarchies, huge packages, very complex code etc.

In this report, we will not distinguish between ***Architectural drift*** and ***Architectural erosion***, since both concepts are normally defined as a gap or divergence between the intended design and the actual code. We will also include both types of the architectural violations: divergences and absences, but consider other common type of software architecture erosion defined by Dalgarno as a part of these two types.

5.3.2 Why do we want to stop software architecture erosion?

The software architecture erosion results in a decrease in the ability of a system's software architecture to meet its stakeholder requirements. Lower quality, increased complexity, harder-to-maintain software, reduced productivity and increased time-to-market are some of the effect of software architecture erosion. In general, architecture erosion leads to a degradation of system quality attributes such as maintainability, reusability and extend ability.

If the erosion has accumulated its effect over long time, it becomes very difficult to understand the *as-intended* software architecture, at the end it could lead to cancelling the project or closing the business. Even though there is a wish to save the software, the only possible option remaining is to rewrite everything from scratch, which is both costly and risky with regards to deadlines or budget. As rewriting the new software will need to achieve all the features the existing software already has and perhaps there are no time left for new features, which are required by the customer right now. It will put both the project and company on stake.

According to Dalgarno in his article *When Good Architecture Goes Bad* [33], the conclusion came from a study by the US Air Force Software Technology Support Centre (STSC) proved that working on a software erosion system not only took twice as long as working on a refactored software system, but also generated eight times number of errors [33].

Another example on issues associated with software erosion is Netscape. After trying to work on a next generation browser based on the old code base for half a year, the developers concluded that the code is eroded beyond repair. They took a big decision to start from scratch. Even during the development of this new browser Mozilla, a lot of code has been rewritten again, since the requirements had changed sufficiently to retire a part of the system before the system was even released. [34]

The last example is the Linux kernel. Like Mozilla, this product is developed as an open source project. It took nearly two years to release version 2.4 after the previous stable release version 2.2. A lot of effort and time was spent on restructuring of the old code in order to meet the new requirements. [34]

5.3.3 Reflection

[Confidential material removed]

6. Why does Software Architectural Erosion happen?

From reading the 108 papers selected by the review process outlined in section 4, we identify seven categories that authors listed as root causes for architectural erosion. ***Table 2 Classification of architectural erosion causes with citations*** shows the mapping between categories, subcategories and the literature references. Each of those categories contains also subcategories, which will be further detailed in the following sections.

Category	Subcategories	Citations
Organization	Deadline pressure	L9, L20, L21, L24, L31, L38, L40, L58, L63, L81, L90, L92
	New requirements: Imprecise/ conflicting	L1, L9, L11, L16, L19, L21, L23, L31, L34, L37, L39, L40, L44, L48, L56, L58, L63, L73, L75, L76, L77, L79, L80, L85, L89
	New hires	L9, L48, L56, L57, L58, L90, L92, L135
	Globally distributed development teams	L72, L75, L136
	Organizational environment.	L58
Knowledge	Limited knowledge	L17, L23, L24, L39, L41, L48, L56, L61, L81, L74, L91
	Knowledge loss	L1, L5, L7, L19, L22, L39, L48, L49, L57, L63, L75, L81
Development process/model	Iterative methods	L42, L46, L81, L87
	Missing methods and tools	L25, L40, L42, L44, L58, L80
Team/Developer	Developer sloppiness	L4, L20, L49, L89
	Misuse and ignorance	L38, L41, L64, L110
	Developers' unawareness	L11, L16, L21, L29, L31, L38, L42, L56, L60, L61, L64, L81
	Developer variability	L58
Coding	Code smells	L32, L63, L74
	Architecture smells	L12, L58, L93
	Poor design decisions	L17, L20, L74

	Complexity of the code and its structure	L17, L49
Documentation	Missing/ Not up to date	L5, L19, L24, L27, L29, L38, L61, L93
	Untraceable design decisions	L5, L22, L33, L46, L61, L63, L64, L69
Others	off-the-shelf (OTS) software	L20

Table 2 Classification of architectural erosion causes with citations

The following figure is the conclusion of H1, it illustrates the classification of the architectural erosion root cause with the categories and their corresponding sub-categories.

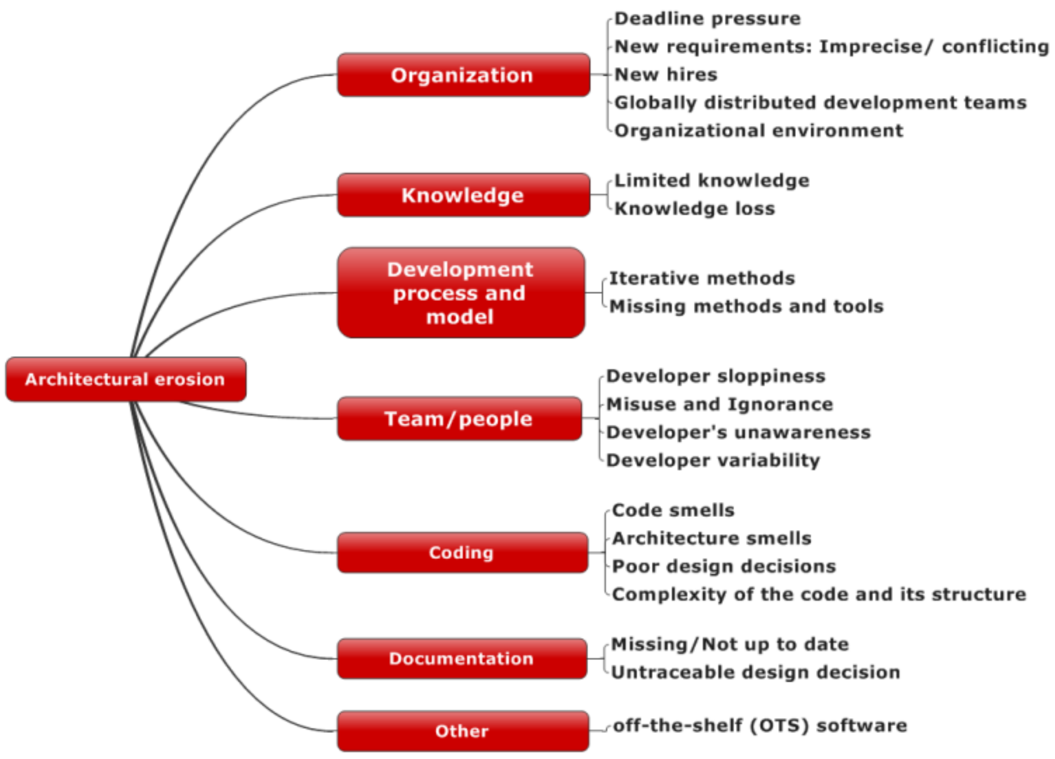


Figure 11 Classification of Architectural erosion root causes

The consequence of all those architectural erosion root causes can reflect into technical debt. Technical debt refers to the consequences of taking shortcuts and workaround when developing software. The term "technical debt" is a metaphor, which was introduced by Cunningham in 1992 [36]. The technical debt arises when developers and architects consciously or unconsciously make wrong technical decisions in order to progress more rapidly in the development of a product. Later, these wrong decisions lead to additional expenses and will delay maintenance and extension. According to Winkler et al. [37] there is a link between architecture erosion and technical debt, which is illustrated in ***Figure 12 Technical debts and architecture erosion***. During the maintenance, the team will add some new technical debt with each change. If the team doesn't think or is allowed to reduce them regularly, the technical debts will increase over time. Technical debts accumulate and with every change, you have to pay the interest on the technical debts. The consequence is, the team will achieve less and less functionality, bug fixes and adaptations to this increasing technical of debt. In the ***Figure 12***, the red arrows are getting shorter and shorter indicating this circumstance.

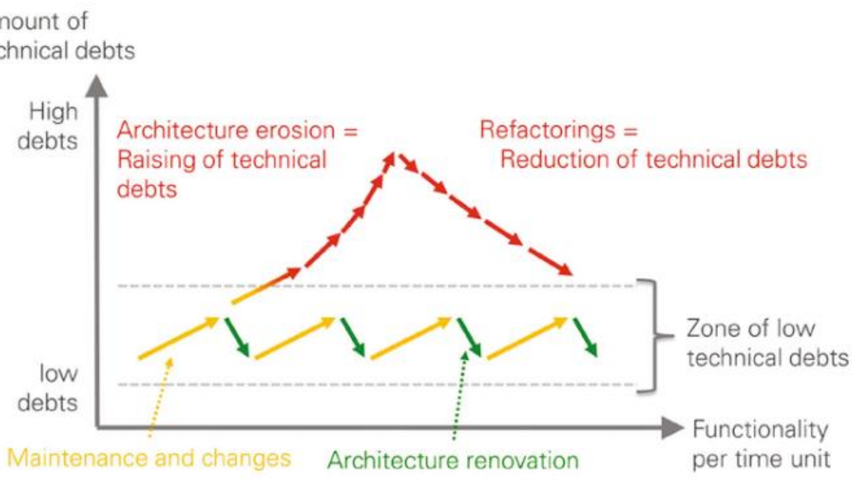

Amount of technical debts
High debts
low debts
Architecture erosion = Raising of technical debts
Refactorings = Reduction of technical debts
Zone of low technical debts
Functionality per time unit
Maintenance and changes
Architecture renovation

Figure 12 Technical debts and architecture erosion

In the following sections, we will provide more details about those architectural erosion root causes.

6.1 Organization

6.1.1 Deadline pressure

According to Gurp et al. [38] project management problems are linked to estimation and planning errors. These problems arise when insufficient time and resources are allocated to a project. Deadline pressures result in not optimal design and sloppy code, because of focusing on reaching the deadline instead of design for change. Deadline pressures can lead developers to take shortcuts and only focus on pure functionality, or make changes without fully understanding their impact on the surrounding system [39]. Parnas [21] also claimed that another impact of deadline is that the developers feel that they do not have time to update the documentation. The documentation becomes increasingly inaccurate thereby making future changes even more difficult.

Deadline pressures can also come from top management, since they are only concerned with the next deadline and they don't consider the future maintenance cost as a top priority.

Another type of deadline pressure is to satisfy the customer's immediate request, some kind of workaround or hack is implemented and incorporated into the architecture. It is often difficult regarding economically and technically to change the fundamental of the architecture to satisfy the new emerging demands. After the software release, there isn't reserved time to do properly implementation, but there is requested for more. The technical debt is accumulated, over time the technical debt gets bigger and bigger and the quality of the architecture degrades because of those violations. At the end, the prescriptive architecture can sometimes not even be recognized anymore in the descriptive architecture.

6.1.2 New requirements: Imprecise/conflicting

Most of the authors agreed that software evolution is the key to architectural erosion. Eick et al. [39] has argued for that imprecise requirements, which can prevent programmers from producing crisp code, cause developers to make an excessive number of changes, which causes code smells.

Other authors claimed the conflicting requirements are the reason for architectural erosion. Eick et al. [39] has also pointed out that requirements load, when heavy, means that the code has extensive functionality and is subject to many constraints. Multiple requirements are hard to understand and the associated functionality is hard to implement.

In addition, a heavy requirement load is likely to have accreted over time and put a both lot of complexities and constraints into the code, which caused the code is doing things it was not designed to do, which results in a big gap between prescriptive architecture and descriptive architecture.

6.1.3 New hires

Ayyaz el al. [40] has claimed that new hires will be a high risk of causing architectural erosion because of lacking enough knowledge about architecture would force them make assumption during modifications to the system.

The same conclusion was also concluded by Eick et al. [39] . Eick et al. also concluded that organizational churn (i.e., turnover or reorganization) increases the risk of decay by degrading the knowledge base and also increase the likelihood of inexperienced developers changing the code. Inexperienced developers can be either new to programming or new to the code or both. They increase the risk of decay because of the lack of knowledge and lack of understanding the system architecture and (for those early in their careers) potential for lower or less-developed skills.

Tvedt et al. [41] argued for that, even the initial design is solid, but with the new people work on the system and if the design of the initial system structure is not followed completely, it will lead to further system degeneration.

6.1.4 Globally distributed development teams

In their study Herold et al. [42] concluded the architectural erosion cannot be avoided completely if the system is large and complex mixing with globally distributed development teams.

In a distributed development, it is normal that developers work on somebody else’s code all the time. It is not only very difficult to get access to detailed information about the software and its underlying rationale, but also because the lack of an explicit connection between the software architecture and the source code, makes it hard for anyone to understand exactly how the software system works and where changes belong. New changes will result into architectural erosion [43].

Messer [44] defined five problems, which the globally distributed development teams need to face all the time:

- We’re thinking ‘us’ versus’ them’: we humans tend to flock together and become ‘us’. The problem here is that, this mindset causes the collaboration to get worse over time. When things go wrong, the two ‘sides’ blame each other.

- Keeping the team in the dark: there are always challenges on knowledge transfer between the teams.

- Culture is a mystery: When the people we collaborate with are from different cultures, things can get worse. We tend to blame things that go wrong, miscommunications and delays, on cultural differences. We don’t understand how people from the other culture think and what drives their behavior. And if we don’t understand the people we need to perform with, it’s hard to collaborate and bring results.

- We stop communicating: when the team is distributed, the team needs more talk, not less, but in reality there is not much communication within the team.

- The black box: The remote team members are far away and we don’t see what they’re doing or even who is working on our projects and what are their priorities. It creates a black box scenario, where we send in requirements or request and just expect the output.

6.1.5 Organizational environment

Eick et al. [39] have also pointed out that the organization environment such as low morale, excessive turnover or inadequate communication among developers can also be the reason for architectural erosion, since these negative factors can produce frustration and sloppy work.

6.2 Knowledge

6.2.1 Limited knowledge

During the maintenance or refactoring because of the limited knowledge of the system, the developers are not able to implement architecturally-relevant strategies for code refactoring and thus wasting time to remove code anomalies that do not represent any threat to the architecture design, but make it worse by introducing new code anomalies [45]. This hypothesis was also agreed by Parnas [21], who claimed that changes made by people that do not understand the original design concept, almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept and therefore will cause further architectural erosion.

Zhang et al. [46] claimed that the reason for developers only having limited knowledge of the system is, because an explicit representation of architectural pattern used in the architecture is missing and when it happens, the modification is made to the system will be conflicts with the initial design.

6.2.2 Knowledge loss

The knowledge is important to be able to keep the architecture intact, but unfortunately the development team usually doesn’t have a constant relationship with the software’s life as there is a possibility that any member can leave the team and the knowledge of the architecture and software associated with him or her also disappears [40]. When those leave the team, new people will normally be added to the project to compensate for the loss. But it takes time for new people to learn the domain and get it up to speed in the project, so mistakes are made and the software erodes further. If new people are unlucky enough to be introduced into a team, where no one knows what the architecture is or should be, then the software will erode even faster as they make changes to it [28].

Bosch has through his studies into design erosion and analysis of its problem, proved that, the key problem is knowledge vaporization [47]. Harrison et al. [48] go further and blame the architects for the knowledge vaporization problem. According to Harrison et al. the knowledge vaporizes because architects fail to record their decisions, so significant information about a software system’s architecture is unavailable during the development and evolution cycles.

In their study Feilkas et al. [32] has identified three manifestations of loss of architectural knowledge: decay of the code in form of violations of the intended architecture, loss of information in the documentation and different perceptions of the prescriptive architecture by different developers. They claimed that the prescriptive architecture was buried as implicit knowledge of the developers and the architects. Therefor between 9% and 19% of all the dependencies implemented in the systems did not conform to the documented architecture. These differences could be identified as insufficiencies in the documentation as well as violations in the implemented architecture.

6.3 Development process and model

6.3.1 Iterative methods

According to the Agile Manifesto [49], working software is valued over comprehensive documentation and responding to change is valued over following a plan. Together with the sometimes “extreme” interpretation with assumption that it means abandoning traditional support activities, many iterative methods activities such as rapid prototyping, extreme programming, etc. are conflicting with the goal of architecture design. The focus and goal of architecture design is to create a plan to move from to implementation in a way that future change requests can be easily covered. Unfortunately, these methodologies typically incorporate new requirements that may have an architectural impact during development without, whereas a careful and optimized design requires knowledge about these requirements in advance [50, 51].

6.3.2 Missing methods and tools

Sergio Miranda et al. [52] mean that, one of the reasons for the architectural erosion is that developers' community lacks tool support for monitoring the implemented architecture. Patzke et al. [53] focusses primarily on software in the product line, but here Patzke et al. also mean that the lack of tools and methods to analyze existing asset bases, make planning how to evolve them a significant problem, which in the long run will cause bugs and mistakes introduced in the system and the software erodes.

Eick et al. [39] points out both *inadequate programming tools*, i.e., unavailability of computer-aided software engineering (CASE) tools and *inadequate change processes*, such as lack of a version control system or inability to handle parallel changes as the causes for architectural erosion.

Other researchers as Dimech et al. [54] claimed that the lack of conformance check process as one of the architectural erosion causes. Because in their opinion, changes in requirements and lack of conformance checks during development are the combination, which can cause the descriptive architecture to deviate from the prescriptive architecture.

6.4 Team/people

6.4.1 Developer sloppiness

A mix of the complexity of the involved systems, the frequency with which they are changed, and the sloppiness, with which the changes are documented, directly contributes to numerous recorded cases of architectural erosion [55, 56]. Because of sloppiness the developer can also choose to ignore the architectural style and decision [57], because it might take longer time to do the "right" things.

6.4.2 Misuse and Ignorance

Changes made by people, who do not understand the original design concept, always cause the structure of the program to degrade [21]. Often the changes will be inconsistent with the original concept and thus invalidate the original concept. Sometimes the damage is small, but often it is quite severe. The problem will start when those changes are in the code and no one can understand the design anymore. To be able to understand the current design, it requires that, one must know both the original design rules and the newly introduced exceptions to the rules. After many such changes, the original designers no longer understand the design. Even those who made the changes, also never did. In other words, nobody understands the modified design.

Medvidovic et al. [57] have also argued for the architectural decisions might also be ignored without justification, due to a missing system-wide view or misguided "creativity" in implementing the desired functionality.

According to Lavallée et al. [58] ignorance problems are linked to a lack of basic knowledge in the field. It is the knowledge related to facts and actual information, sometimes called "declarative" knowledge. For example, not knowing the content of an API would be an ignorance problem. But misuse problems are linked to a lack of experience in the field. This is the knowledge linking performed actions with desired goals, sometimes called "procedural" knowledge or knowhow. For example, the abuse of the static keyword would be a misuse problem, as the developers are not conscious of the link between making an object static and its impact on encapsulation. Gurp et al. [59] also identified this problem by concluding that poor programming techniques led to poor code quality in a highly flexible programming language.

6.4.3 Developers’ unawareness

Lavallée et al. [58] define the difference between ignorance and unawareness is that developers are aware of the basic concepts manipulated, but are unaware of how to integrate them into their current context. In their study Gurp et al. [59] has identified that the problem with lack of process awareness resulted in undocumented changes, which in turn led to inconsistency problems.

6.4.4 Developer variability

Eick et al. [39] means that developer variability in a team can also be a reason for architectural erosion. Because their more skilled colleagues had written complex code or used architectural style and design pattern, which he/she didn’t understand. He/she will need to make some assumptions, which would introduce the violations into the architecture.

6.5 Coding

6.5.1 Code smells

Arcoverde et al. [60] define code smells as an indicative of structural problems and design problems, which make code harder to read and maintain. They can be even more harmful when they impact negatively on the software architecture design, which over time will cause architectural erosion. Blob and Spaghetti Code are examples of code smell [58]. Mäntylä et al. [61] and Schumacher et al. [62] map duplicate code, god class, and long parameter list as typical code smell, which causes architectural erosion.

6.5.2 Architecture smells

Architecture smells are combinations of architectural constructs that reduce system maintainability. Architectural smells are analogous to code smells because they both represent common “solutions” that are not necessarily faulty or errant, but still negatively impact software quality. An inappropriate architecture that does not support the changes or abstractions required of the system [39] will only work if the system doesn’t need to evolve. But a system without evolution, will age [21] to dead. The most important word in a long-lived system is “evolution”. No architecture is a onetime shot. Software will always be changed and adapted to new requirements and new end-user needs [63]. In other words, a system without an adaptable architecture will erode sooner than a system based on an architecture that takes change into account.

Martin [64] presented seven symptoms of poor architecture:

- **Rigidity**: this means the system is hard to change. Every change forces other changes to be made. The more modules that must be changed, the more rigid the architecture.

- **Fragility**: when a change is made to the system, bugs appear in places that have no relationship to the part that was changed.

- **Immobility**: this is when a component cannot be easily extracted from a system, making it unable to be reused in other systems.

- **Viscosity**: this is when the architecture of the software is hard to preserve. Doing the right thing is harder than doing the wrong thing.

- **Needless complexity**: the architecture contains infrastructure that adds no direct benefit. It is tempting to try to prepare for any contingency, but preparing for too many contingencies makes the software more complex and harder to understand.

- **Needless repetition**: this is when architecture contains code structures that are repeated, usually by cut and paste, that instead should be unified under a single abstraction.

- **Opacity**: this is when the source code is hard to read and understand.

Ambiguous Interface and Scattered parasitic functionality are examples of architecture smell [65].

6.5.3 Poor design decisions

Design decisions are hierarchical in nature. A high-level architectural decision is followed by many low-level decisions, and design decisions are accumulated and interact in a way that, revision of one would force reconsideration of all the others. A consequence of this problem is that if developer makes design decisions, which has impact on the architectural level, it will cause a chain reaction and all other decisions need to be revised. According to Jaktman et al. [66] erosion can be a result of poor design decisions made while implementing maintenance changes to the system.

Arcoverde et al. [60] means that decision making problems are linked to team dynamics. These problems arise when design decisions are not sufficiently evaluated and debated. Major decisions made in an unconcerned way or without accounting for a ripple effect can lead to a broken structure. A lack of impact analysis or impact monitoring can also lead to harmful decisions.

6.5.4 Complexity of the code and its structure

Jaktman et al. [66] also claimed that due to complexity of the code and its structure, the system can be hard to understand and maintain. It will lead to architectural erosion during maintenance.

6.6 Documentation

6.6.1 Missing/ Not up to date

Developers maintaining a product are often not involved in the initial design. In these cases, missing documentation or the lack of up to date documentation complicates product maintenance and extensibility [63].

We all know that software architecture documentation is essential for preventing architecture erosion as a major concern of sustainable software systems. However, the high effort for elaboration and maintenance of architecture documentation hinder its acceptance in practice especially in the industries [67]. Another issue is that when documentation is written, it is usually poorly organized, incomplete and imprecise. Often the coverage is random or developers are not sure what to document. Often a developer or manager decides that a particular idea is clever or important and writes a memo about it while other topics, equally important, are ignored. In other situations, where documentation is a contractual requirement, a technical writer, who does not understand the system, is hired to write the documentation. The resulting documentation is ignored by the maintenance programmers because it is not accurate. Worst case scenario is that some projects keep two sets of documentation; there is the official documentation, written as required for the contract, and the real documentation, written informally when specific issues arise [21].

6.6.2 Untraceable design decisions

Not being able to trace design decisions is one of the main reasons for software architecture erosion. High level architectural decisions are difficult to trace at the source code level [68, 59]. Therefore, a developer working on a new feature might fail to recognize the original architectural decisions and thus fail to make sure those architectural decisions are upheld.

Jan Bosch [47] means that problem with design decisions is that the architecture design decisions lack a first-class representation in the software architecture. Once several design decisions are taken, the effect of individual decisions is implicitly present, but almost impossible to identify in the resulting architecture. A consequence of this problem is that knowledge about the “what and how” of the software architecture is quickly lost. Some architecture design methods stress the importance of documenting architecture design decisions, but experience shows that this documentation often is difficult to interpret and use by individuals not involved in the initial design of the system.

According to Harrison et al. [48] there are two issues about documentation design decisions. One is that it is difficult to get people to record not only the decision itself, but also recording the critical information surrounding a decision. Another issue is that architects often fail to adequately document their decisions because they don’t appreciate the benefits, don’t know how to document decisions, or don’t recognize that they’re making decisions. This lack of thorough documentation can significantly disrupt the system when decisions made later, during subsequent development iterations, conflict with the original architectural decisions. Gerdes et al. [67] also mean that there not only a need for design decision templates, but also an architecture knowledge management.

Whereas Zhang et al. [46] focus on documenting the “right” design decisions. They argued for that there are too many design decisions but not all of them are that important, and it is expensive to document design decisions. Therefore, they proposed an approach to only capture the minimal set of the really important ones.

6.7 Others

6.7.1 Off-the-shelf (OTS) software

Medvidovic et al. [57] means that using OTS software can also result into architectural erosion in the later development phase. The reason for that is that the OST’s functionality often is directly incorporated into the system’s implementation; and the existence of legacy code that is perceived to prevent careful system architecting.

6.8 Reflection

6.8.1 AK2 case study

[Confidential material removed]

6.8.2 Discussion

It is generally common opinions that the architecture is eroded during the maintenance activities. Because during the maintenance new requirements and new technology are introduced, which often contort or violate existing design structures in unplanned ways. This in turn creates structures that are even more difficult to understand and maintain, which contributes to the increased complexity of the source code.

[Confidential material removed]

Even though all the reasons for architectural erosion are classified in different categories, but most of them is a trigger to another. For example because the new team member doesn’t know about the code and architecture, the ***new hires*** will cause the ***limited knowledge*** and ***developer variability***. The same if the organizational environment is bad, it will cause ***developers sloppiness*** and ***misuse and ignorance***. All the causes will reflect in the code in the descriptive architecture. ***Figure 13*** illustrates the relations between the subcategories.

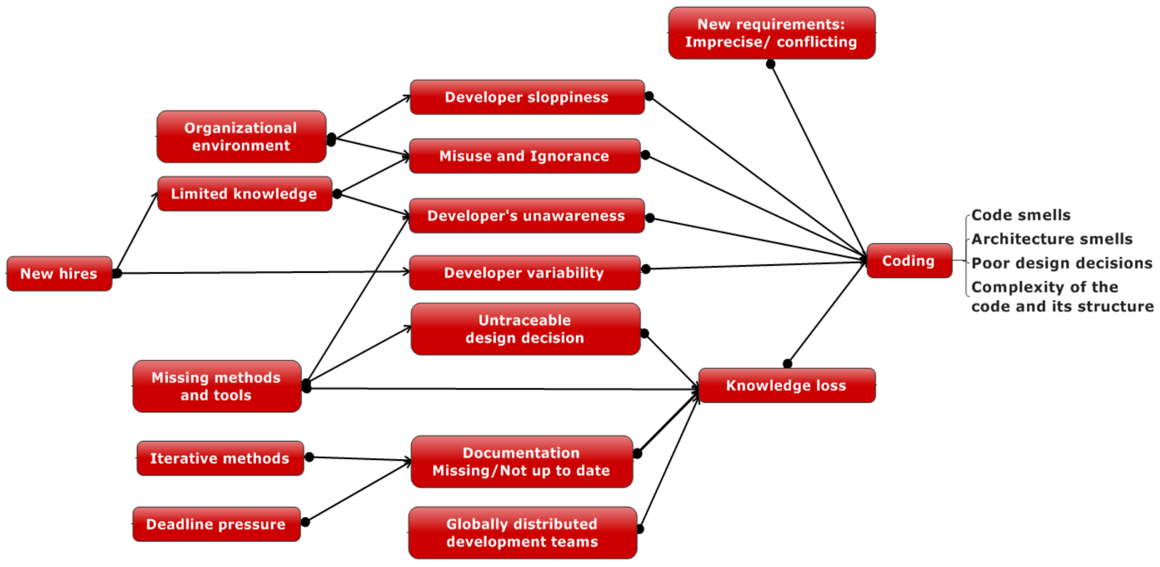

New requirements: Imprecise/ conflicting
Developer sloppiness
Organizational environment
Misuse and Ignorance
Limited knowledge
Developer's unawareness
New hires
Developer variability
Coding
Code smells
Architecture smells
Poor design decisions
Complexity of the code and its structure
Untraceable design decision
Missing methods and tools
Knowledge loss
Iterative methods
Documentation Missing/Not up to date
Deadline pressure
Globally distributed development teams

Figure 13 Relations between the architectural erosion causes

As many authors proved that the architectural erosion is not avoidable, so the only way to detect and take some action to prevent or minimize the damage. In the next section, we will present the first step: detecting. We will report and classify the existing techniques for detecting the architectural erosion.

7. How to detect Architectural Erosion?

Early architectural erosion identification is desirable so that steps can be taken to prevent further erosion. The question is how to identify this decay and what to do, to stop it. In following sections, we will present the techniques used to detect architectural erosion, which we have extracted from the 108 selected papers. ***Table 3*** gives the summarized view of the different strategies to detect architectural erosion. The techniques can be categorized as manual and semi-automated approaches.

Detecting techniques	Category	Subcategory	Citations
Manual approach	Inspection of source code	Code smells	L90, L92
	Inspection of architectural artifacts	Architecture smells	L94
Semi-automated approach	Metric-based	Code smells	L95, L96, L97
	Historical data analysis	Change management data	L58, L69
		Architecture history	L25, L105
		Defect-fix history	L106, L107, L108
	Compliance checking	Reflexion model	L33, L50, L53, L91, L84, L98, L99, L100, L101, L105
		Dependency rules	L11, L30, L57, L102, L103, L104

Table 3 Architectural erosion detection techniques with citations

The architectural erosion detection techniques, which is also **conclusion of H3** is visualized in ***Figure 14***. We will provide more detail for the single technique in the following sections.

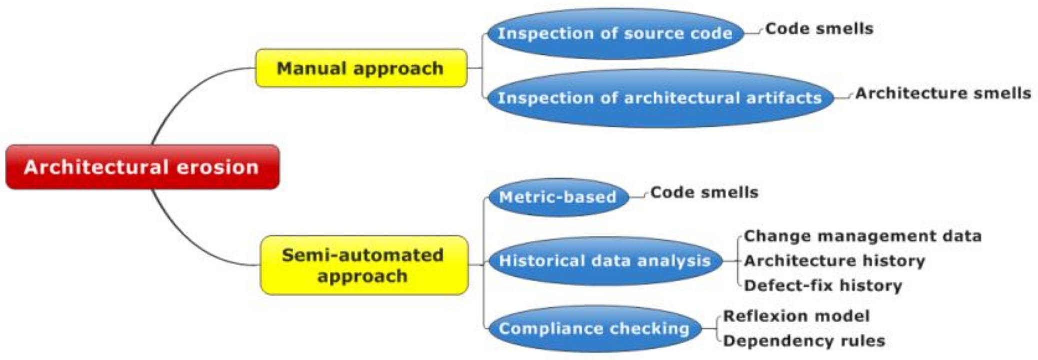


Figure 14 Detection techniques

7.1 Manual approach

7.1.1 Inspection of source code

Detection techniques consist of manual visual inspection of source code and architectural artifacts. Source code inspections are performed by developers and are guided by questionnaires. Mäntylä et al. [61] presented the technique by asking developers inspected source code to identify code smells. By filling out a web-based questionnaire and performing an assessment based on subjective evaluation on a seven-point numeric Likert scale, Mäntylä et al. could identify three different code smells: duplicate code, god class, and long parameter list. These found code smells do not correlate with code smells found using automated metrics tools.

Schumacher et al. [62] focus more on the relation between god class and maintainability. In their study, the developers were encouraged to think aloud as they filled out the questionnaires. Schumacher et al. compared the subjective results with the metric values from the automated metric-based tool. The automated metric-based tool was only able to detect the god class with a precision of 71%. Adjusting the thresholds of metrics used in the detection strategy allowed to increase metric-based tool's precision to 100%. The results of their study increased the overall confidence in using metric-based tool' to detect code smells.

Travassos et al. [71] also presented a process based on manual inspections and reading techniques to identify smells. The process only covers the manual detection of smells, not their specification.

7.1.2 Inspection of Architectural Artifacts

Inspection of architecture artifacts is done by subjective evaluations using checklists and by comparing architecture models. Bouwers et al. [72] proposed the lightweight sanity check for implemented architectures (LiSCIA) to identify architectural erosion. Developers evaluate implemented architectures by inspecting the architecture artifacts. The evaluation phase consists of answering a list of 28 questions based on units of modules, module functionality, module size, and module dependencies. Based on the corresponding actions to the questions provided by LiSCIA, the developers could identify the erosion in an implemented architecture.

7.2 Semi-automated approach

7.2.1 Metric-based

Lanza et al. [73] argue that the manual detection of code smells is time consuming, non-repeatable and does not scale. They proposed a metric-based approach to detect code smells automatically. Marinescu went further and defined detection strategies [74], which combines different code metrics and information, then filters the result to detect problems in the code structure. Marinescu has concluded in his study that the precision of the automatic detection is reported to be 70% comparing to detection performed by a human. Later in another study, Munro et al. [75] refined Marinescu's method by suggesting a more systematic approach. This includes providing empirical evidence for choosing the software metrics used for the automatic detection. They also propose new metrics based on the characteristics and design heuristics associated with the code smell.

7.2.2 Historical data analysis

7.2.2.1 Change management data

Eick et al. [39] were using the history of change management history combining with a conceptual model of code decay, code decay indices and statistical analyses to detect code decay. Change management history includes source code of the feature, modification request, delta, and change to severity levels. The statistical analysis on those data showed that an increase in the number of files touched per change and decrease in modularity over time yields strong evidence of code decay.

Stringfellow et al. [76] claimed that change management data is useful in measuring properties of software changes and such measures can be used in making inferences about cost and change quality in software production. They proposed to use change reports as a major source of information to identify erosion. According to their study, change reports are written when developers modify code during system development or maintenance and usually showed the coupling between components. By analyzing those couplings, problems and possible architectural erosion can be detected via changes related to components and interactions of the components

7.2.2.2 Architecture history

Brunet et al. [77] define architectural debt as the difference between the amounts of solved and introduced violation. By analyzing the violations' lifecycle for several versions of four different open source systems, they could compute those architectural debts. They were also able to identify not only the method that causes the violation, but also its class, package and module, once this hierarchy is defined in the high-level module view.

Hassaine et al. [26] used the architectural histories of three open source system in their study to detect architectural erosion in software evolution. The architecture histories consist of architectural diagrams of different versions that are extracted from source code using a tool. In their approach ADvISE, they performed pair-wise matching of the subsequent architectures to identify deviations in the actual architecture from the original architecture by tracking the number of common triplets. These triplets (S, R, T) represented the extracted architecture, where S and T are two classes and R is the relationship between two classes.

7.2.2.3 Defect-fix history

Li et al. [78] used architectural defects and fix relationship history to measure architectural erosion. They defined architectural defects as the defects, which were spanning in multiple system components and fix relationships as fixing one component requires fixes in other components due to architectural defects. Architectural defects and fix relationship history consists of information about the release, component in which the defect occurred, and the number of files changed to fix the defect. They analyzed the histories by applying multiple component defect metrics for example percentage of defects that affected multiple components in a system and the average quantity of files changes to fix a defect. They concluded that if these metrics value increased between two versions of the system, it would indicate that the architectural erosion has occurred.

Similarly, Ohlsson et al. [79] used simple coupling measures based on common code fixes as part of the same defect report for analyzing and detecting code decay. They were able to identify the most problematic components across successive release. They argued that the increasing number of changes, size, effort and coupling metrics would indicate the architectural erosion. Mayrhauser et al. [80] focused on the most fault-prone components and component relations to detect the code decay. They used a defect coupling measure that combined the number of files that had to be fixed in each component for a specific defect. These defects cohesion and coupling measure were computed for the most fault-prone and components relations that contain fault, since they represent the worst problems and the biggest potential for code decay.

7.2.3 Compliance checking

7.2.3.1 Reflexion model

Besides Dependency Rules, Reflexion model is one of the most popular detection methods. We will present more details about the Dependency Rules in the next section. Reflexion models were originally introduced by Murphy el al. [81] to check the difference between the higher-level design of software systems and their implementations in a light-weight and iterative manner. Later Koschke et al. [82] extended the model to support hierarchies. Similarly, Herold et al. [83] proposed a more feature oriented approach, where they combined feature location and the reflextion models.

Figure 15 Reflexion Modelling shows the six steps techniques that are the basic of reflexion model. The technique helps developers to improve their comprehension of software systems and supporting architects and developers in their effort to keep software architecture and code consistent.
The Reflexion model’s six steps consist of:

1. Creation & refining step: Before implementation of the system commences, the developer creates a hypothesized architectural model, the prescriptive architecture.
2. Creating mapping: During the implementation phase, the developer would create a mapping between the prescriptive architecture and the implementation.
3. At a point during implementation or subsequent maintenance, a dependency graph of the system’s sources can be extracted by parsing the system, creating the descriptive model.
4. Evaluation step: The relationships defined by the developer in the prescriptive architecture are compared with those extracted from the descriptive architecture. Results of that comparison are presented to the developer through the Reflexion model periodically.

5. Refining step: By analyzing particularly the inconsistent relationships and violations in the Reflexion model, the developer chooses to either alter the mappings, updating the code base or updating the prescriptive architecture model.
6. Recursive step: The steps 2-5 are continuously repeated over time, towards prompting increased system conformance to the prescriptive architecture.

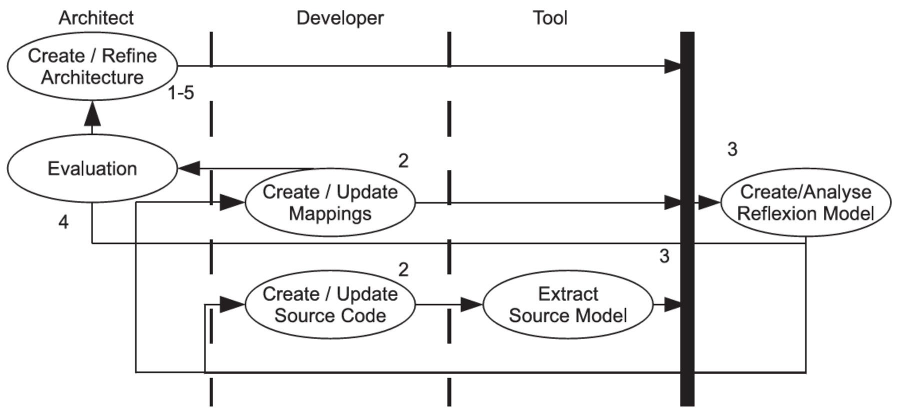

Architect
Developer
Tool
Create / Refine Architecture
1-5
Evaluation
4
2
Create / Update Mappings
3
Create/Analyse Reflexion Model
3
2
Create / Update Source Code
Extract Source Model

Figure 15 Reflexion Modelling

When the Reflexion Model is available, it can be used to detect the architectural violations.

7.2.3.2 Dependency rules

Dependency analysis can expose violations of certain architectural constraints such as inter-module communication rules in a layered architecture, thus identifying potential instances of erosion. Terra et al. [84] presented a domain-specific dependency constraint language that allows software architects to restrict the spectrum of structural dependencies, which can be established in object-oriented systems. Dependency Constraint Language is a domain-specific language with a simple, easily understandable syntax for defining structural constraints between modules. Once defined, such restrictions are statically checked by a conformance tool, called dclcheck. ***Figure 16*** illustrated the approach, which relies on static analysis techniques to detect structural dependencies that are indicators of architectural erosion. The architects must initially specify a set of structural constraints for the target system, using the dependency constraint language.

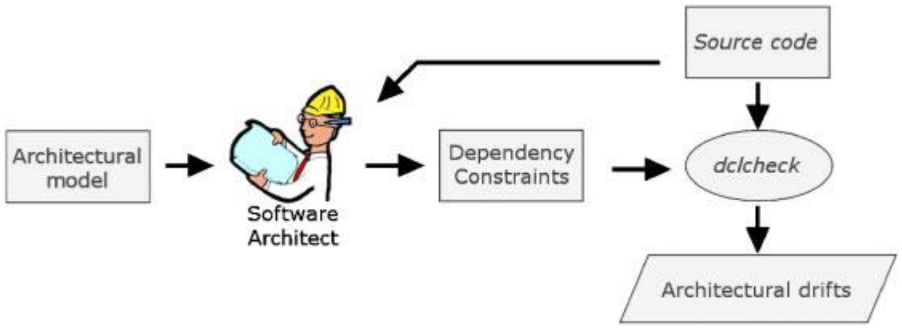


Figure 16 Architecture conformance dependency constraint language

Pruijt et al. [85] proposed that the architecture should be define as semantically rich modular architecture (SRMA), which is expressive and may contain modules with different semantics, like layers and subsystems, constrained by rules of different types. According to Pruijt et al, SRMA was able to provide a language to express constraints on the modules in an architectural mode. In their study, Pruijt et al. used an architecture compliance checking tool (dclcheck) to analyze the code and detect the architectural drifts. Sangal et al. [86] also proposed to extract dependencies from the code and show them using a scheme that highlights potential problems. The developers still needed to enter 'design rules' that capture the architect's intent about which dependencies are acceptable. The design rules are applied repeatedly as the system evolves, to identify violations, and keep the code and its architecture in conformance with one another.

Tvedt et al. [41] claimed that visual inspection of the architecture might not be systematic enough to detect deviations. They proposed an approach to detect and avoid architectural erosion by actively and systematically detecting and correcting deviations from the planned design as soon as possible based on analysis of couplings between components.

7.3 Reflection

The detection strategies we found in this review are categorized into manual and semi-automated approaches. Manual inspection of source code or architectural artifacts is tedious work. Moreover, this process is time consuming, non-repeatable, and non-scalable, especially for the larger system, thus it is expensive.

[Confidential material removed]

The semi-automated approaches involve less human intervention, but there is still a human based task. The historical data analysis is useful only if the data is available, informative and consistent, which is not in the AK2 system case. The metric based approach is helpful in identifying structural violations in design patterns and architectural styles. But it requires a fine definition of the metrics with threshold filtering rules. The disadvantage of this technique is threshold values are determined by developers' opinion, which might not possible to apply to all the systems uniformly.

The reflexion models are clearly recommended as it was mostly used in a lot of studies. Reflexion model seems to provide a clearly defined architecture conformance process. But reflexion models not only require a mapping between the prescriptive architecture and the implementation, but also successive refinements in the high-level model to reveal the whole spectrum of absences and divergences that can be present in the source code [82, 87]. This mapping and refinements process requires a big effort, especially as it is needed to be maintained over time. But once the architectural model and the mapping are in place developers could use the tool to make sure that no violations of the architecture takes place. Unfortunately, studies show that developers still heavily rely on manual reviews to check architectural conformance. In fact, static analyzers are used in only 33% of the cases [88]. The use of manual techniques does not scale and entails additional costs that could be minimized by using existing technologies and tools. On other hand, these tools can generate false positive warnings, as common in most static analysis tools, which will generate a feeling of waste of time for developers.

We always have an assumption that, if we can find the points in the evolution of software where high-impact erosion is introduced, it could also be possible to fix this erosion. But in reality, even if the erosion is found, it is not always possible to remove the erosion due to the complexity of wide spread erosion and time constraints [89]. Therefor it is a need to detect the architectural erosion as early as possible to be able to stop it or at least limit its damaged effect.

Detecting and representing architectural violations in a reflexion model are only the first two steps towards resolving such violations and towards reversing the architecture erosion they represent. After that, violations have to be analyzed to understand what causes them in order to take effective actions to resolve the violations. The result of the analysis report is invaluable and important knowledge. Because the knowledge of variability evolution does not only help to understand erosion in the past, but also helps to identify prevalent erosion trends in the future.

8. Avoiding the architectural erosion

In the following sections, we will present the techniques to avoid the architectural erosion. We perform a survey of techniques for avoiding architecture erosion and their advantages and limitations. We are using the same categories classification from ***Section 6*** to classify existing approaches for avoiding architectural erosion. Each of these categories contains one or more sub-categories based on the high-level strategies used to realize its goal. The following figure is illustrated the approaches for avoiding architectural erosion, more details about those approaches are presented in the following sections.

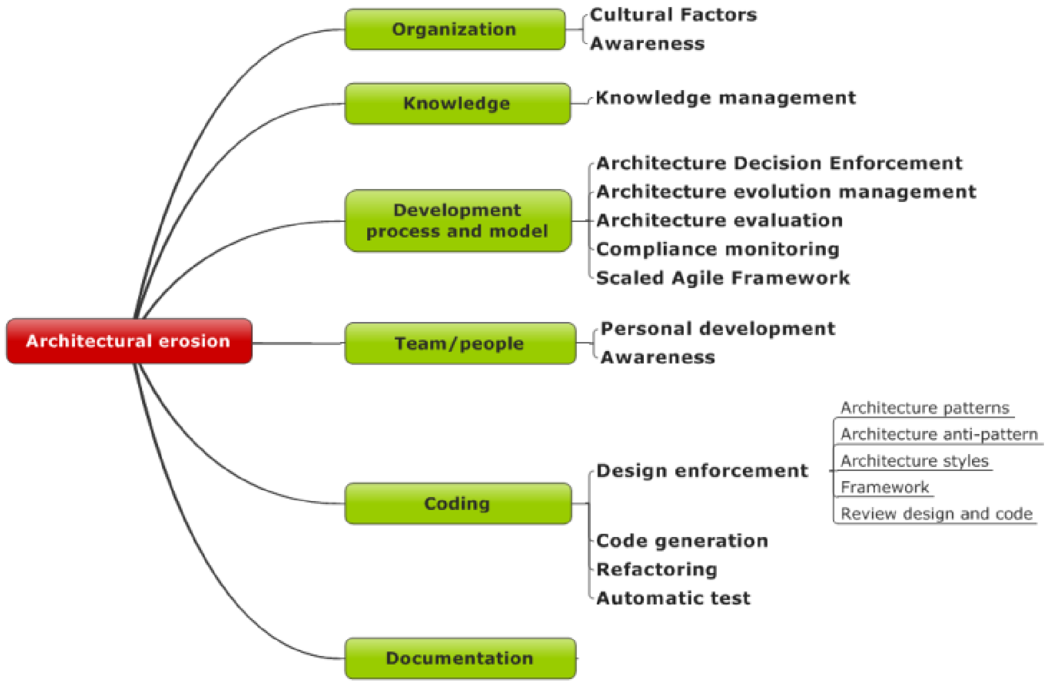


Figure 17 Avoiding architectural erosion approaches

The avoiding approach contains those strategies which are targeted towards avoiding or at least limiting the occurrence and impact of architecture erosion but may not be effective in eliminating it. The first step of avoiding the architectural erosion is to increase architectural erosion awareness within the organization. Because to be able to take any action against architectural erosion, we need to know what the symptoms are for architectural erosion. The goal here is to notice any signs of architectural erosion as early as possible. These symptoms can also be considered as consequences of erosion and they can be easily spotted without any big analysis effort.

- **Inflexibility**: code is hard to understand, makes the software difficult to change as a change can cause violation in dependent modules. No one can predict the impact of the change. In this state, the manager will eventually refuse any changes in the software because of the fear of side effects by modification or changes [40, 90].

- **Serenity**: unable to reuse components from same or different software projects as most of the software involves much similar type of modules written by other developers. Refactoring the code will raise the question about whether it is better to rewritten instead of reuse [40].
- **Increased cost:** decrease in productivity and quality dues to defects. Increase in time, complexity, effort and risk for implementing and test new functionality [90].

With those symptoms in mind, we can become more awareness of the architectural erosion, thus taking action to avoid and minimize the erosion.

8.1 Organization

8.1.1 Cultural Factors

Messer [44] proposed an approach to fight against the cultural problems, which normally occurred in a ***globally distributed development team***. The approach divided into three areas:

- Culture & team spirit: we need to consciously work on team spirit and we need to address cultural differences, so we can organize around them and build trust over distance.

- Knowledge transfer: the goal here to ensure that all the team members have equal access to the information. This relates to ‘talk time’ to the product owner, feedback from users and detailed product information (roadmaps, documentation, vision).

- Communication rhythm: establish a communication rhythm, meet often and meet well. People don’t like meeting, thus it is important to keep only the meetings, which introduce value to the team and the project. For example: the daily and the retrospective. The daily serves to create alignment between the team members and a chance to remove impediments. The retrospective serves to improve the way we work continuously. It’s our opportunity to voice concerns about our project, our team, and the way we collaborate.

8.1.2 Awareness

Even though the value of an improved architecture and the consequences of an eroded architecture are clear to technical staff, but it is often difficult to convince upper management that the extra effort is necessary. Improved architecture is intangible and does not translate into visible user features that can be marketed [91]. Consequences of an eroded architecture is hard to translate to financial numbers and only visible if the technical staff are self-aware of them. The fight against with software erosion requires management commitment. If managers are only interested in the short-term viability of their software projects then it is hard for developers to get the time and make the effort to tackle the problem [28].

Therefore, there is a need of increasing awareness of architecture erosion within the organization. Management obligation is very important for fighting architecture erosion otherwise it would become very much difficult for developer to deal with the problem in a timely manner. With management support, a culture, where fighting against erosion is valued, can be created. This culture is likely to have characteristics such as – an emphasis on regular refactoring, clear assignment of responsibilities, sharing of architectural knowledge and work, frequent communication between the whole groups [28]. The managers will need to handle carefully when a project is transferred or handed over to another team or a new team member joins the team. Every new developer need both time and help to get up to speed. Management should also create a supportive and motivating environment where appropriate training should be provided to new hires so that they must understand the system and senior employees would not be overburdened thus avoiding turnover and reducing the risk that the architectural knowledge would be lost with the person leaving the organization.

Gilb argued for that the team needs to put time aside to focus on improving their software's maintainability. It had the added benefit of making the development team feels empowered [92].

As mentioned in the ***section 6.1*** one of the causes of the architectural erosion is organization environment, which also requires management's attention.

8.2 Knowledge

8.2.1 Knowledge management

Developers need knowledge about the prescriptive architecture of a system, whenever they do any modification. Without this knowledge, the developers can break the architectural integrity of the system accidentally, even by making only small code changes. To fight against knowledge lost, it not only requires well-documented architecture as discussed in section 5.2, but also knowledge management. Ali Babar et al. [93] draw a distinction between personalization strategy and strategy for architectural knowledge management, where personalization strategy emphasizes on the interaction and communication among developers, and architectural knowledge management concentrates on identifying and storing knowledge in artifacts and repositories.

Feilkas et al. [32] has concluded in their study that in order to minimize the knowledge loss, we need to make the knowledge about the intended architecture explicit and perform automatic architecture conformance analyzes continuously in order to keep the awareness of developers about the architectural knowledge.

8.3 Team/people

8.3.1 Personal Development

One way to avoid the architecture is to ensure that the developers share a good understanding of the system's architecture [94], which means that the developer need to have the basic skill to both understand and implement the architectural patterns, design patterns and avoiding anti-pattern. Therefor further education of the developers is an important factor to avoid architectural erosion. Both Schrøder et al. [95] and Christensen et al. [18] claimed that *"People quality is as important as structure quality"*. They meant that good communication and collaboration skill was highly important for the architect's role to be able to present the architecture idea to its stakeholders. Fairbanks [96] emphasizes that knowing the features of the programming language does not mean you can design a good object-oriented system, nor does knowing the Unified Modeling Language (UML) imply you can design a good system architecture. It requires focusing on the technology of architecture and those supporting tools around it, rather than organizational roles or processes.

Parnas [21] claimed that, undereducated about the tools and processes are also the reason for architectural erosion. For example lack of training in design reviews, where many developers do not know how to prepare and hold a design review. Or lack of training in design documentation, where developer don't know how to produce a well-documented design and architecture. Or lack of training in specification documentation, those who need to write specification don't know how to produce a precise and measureable specification.

8.3.2 Awareness

As mentioned in section 8.1.2 awareness plays a big role in avoiding architectural erosion within an organization, but awareness within team and people will also help to detect and avoid architectural erosion early. In their study Gurp et al [38] proposed to involve different people in diagnosing an eroding system and in evolving it. Normally maintenance and repairing the system was not taken by the original developers of the system, but by people who had inherited the system from their predecessors. The study showed that by involving different people, it is not only increasing awareness among the developers, but also ensuring developers' good understanding of the system.

8.4 Development process and model

8.4.1 Architecture decision enforcement

During the design phase, developers must make decisions and reify them in the code. The decisions made during software architecting are particularly significant in that they have system wide implications, especially on the quality attributes. To ensure the correct implementation of decisions in the source code, Zimmermann et al. [97] proposed a model-driven approach called decision injection. Jensen et al. [98] supported the approach by implementing a tool that allows the management of the architectural decision. They argued that, because of the architectural decisions often aren't explicitly documented, and therefore they are eventually lost. Consequence of the loss are; it created major problems such as high-cost system evolution, stakeholder's miss-communication, and limited reusability of core system assets. Their tool is an attempt to bind architectural decisions, models and the system implementation. One of the tool features is that it provides support for checking the implementation against architectural decisions. One of the issues in this approach is that, there is not clear what type of decision and decision violations, which has impact in architectural erosion and need to be handled.

8.4.2 Architecture evolution management

This approach is to minimize architectural erosion by managing changes to both the implementation and the architecture in parallel while ensuring they are consistent to each other during evolution process. Evolution management is performed with the help of software configuration management (SCM) [99] tools, also known as version control systems. By monitoring and controlling changes to an architecture specification and its implementation artefacts, it can provide the ability to validate the implementation against the architecture, after a change has been made to either artefact. The issue with this approach is that the SCM tool has to be made aware of the architectural rules, which is a mapping of the prescriptive architecture and descriptive architecture. It added both dependency and complexity of the evolution process.

To ensure a consistent evolution process and make it easier to add new functionality to the system, Lindvall et al. [91] has defined the maintenance process, which can be broken down into following sub processes:

a. Understanding the change request
b. Understanding the system and its structure
c. Localizing where to change the system in order to implement the change request.
d. Implementing the primary changes
e. Determining the ripple effects
f. Implementing the secondary changes
g. Testing that the system fulfills all previous as well as new requirements

8.4.3 Continuous architecture analysis

As mentioned in ***section 5.2.2***, Architecture Tradeoff Analysis Method (ATAM) can be used to evaluate the prescriptive architecture with respect to user requirements, quality attributes, design decision, design tradeoff and associate priorities and risk levels with important scenarios. The outcome of an ATAM analysis is a utility tree that presents a comprehensible view of the risks, sensitivity points and non-risks in the architecture. A number of other architecture analysis methods have been developed including the Scenario-based Architecture Analysis Method [100] and the Software Architecture Evaluation Model [101], all are recommended to made an evaluation of the architecture as early as possible, before the design is complete as the definition of architecture and design decision has a strong influence on the final quality of the product. But Lindvall et al. [91] suggest that the architecture evaluation can be conducted at different points in time during the software cycle because their goals are different. They distinguish between early and late architecture evaluation. Early architecture's goal is to evaluate one or more software architecture candidates that are not implemented yet to ensure better understanding of the architecture and to choose a right candidate. Their focuses are on:

- Verification that all requirements are accounted for
- Indication that the system will have the desired qualities or quality attributes (e.g. performance, reliability, and maintainability)
- Identification of problems with the architecture.

While late architecture evaluation is used to evaluate the software architecture of an implemented version of a system compared to the software architecture of the previous version. The result can be used to evaluate whether the new actual software architecture fulfills the planned software architecture and whether it better fulfills the defined goals and evaluation criteria than does the previous actual software architecture.

Feilkas et al. [32] proposed a continuously architecture analysis to ensure that the architecture description will be kept consistent with the code as a specification of the prescriptive architecture.

8.4.4 Compliance monitoring

As mentioned in ***section 7.2.3***, conformance checking is another enforcement method; it allows the enforcement of the modular structure and dependencies of the software system. A software process that includes regular compliance monitoring activities can keep the implementation faithful to the intended architecture as the system develops.

Gurp et al. [38] proposed monitoring defect metrics. They argued for that the symptoms of design erosion are the defect rate and the average time needed to fix these defects. By monitoring these metrics, it may help identify problems with respect to eroding designs at an early stage or at least serve as an early warning system for problematic systems. This approach is unsurprisingly cheap, since the metrics are relatively easy to collect and with an assistance of defect reporting system, the precise location of the defects can be discovered.

8.4.5 Scaled Agile Framework

Scaled Agile Framework (SAFe) introduced ***architectural runway*** as a tool to implement the agile architecture strategy. The ***architectural runway*** consists of the existing code, components and technical infrastructure needed to implement near-term features without excessive redesign, and delay [102]. The architect defines ***enabler***, which promote the activities needed to extend the architectural runway to support future business functionality. These include exploration, infrastructure, compliance, and architecture development. By using ***enabler*** to build up the runway, they make sure that there is enough attention on the architecture activities. ***Figure 18*** illustrated how the architectural runway continuously evolves to support new features.

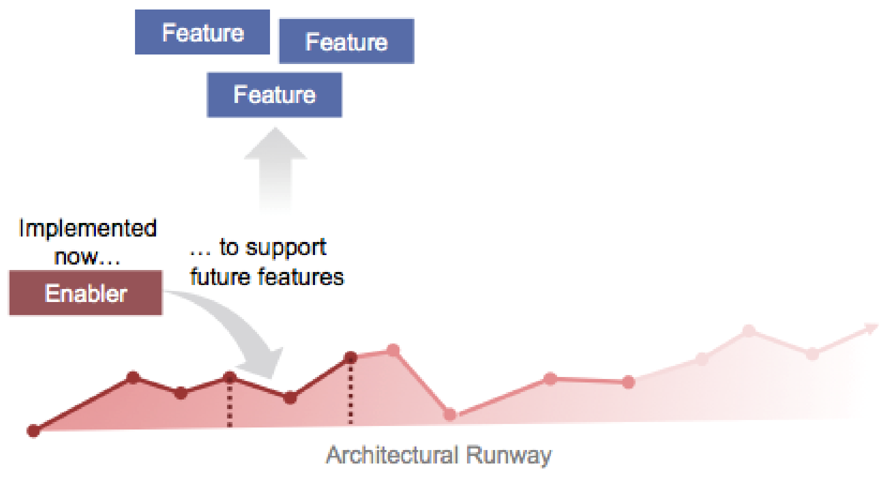


Figure 18 The architectural runway continuously evolves to support future functionality

To avoid producing redundant and/or conflicting designs and implementations, the team need intentional architecture. The intentional architecture is a set of architectural guidelines that enhance solution design, performance, and usability and direction for cross-team design while syncing implementation.

8.5 Coding

8.5.1 Design enforcement

In the following sections, we will present the design enforcement approaches. Normally those approaches are established at the beginning of the implementation process, where the architectural model is transformed directly into source code or UML models. The prescriptive architecture is refined using architecture patterns, architecture style, architecture anti-pattern or framework to guide and impose certain constraints on the implementation.

8.5.1.1 Architecture patterns

The architecture patterns are well-established and documented solutions to recurring architectural design problems. Architectural pattern expresses a fundamental structural organization or schema for software architecture design and in fact it is incorporated in most software architectures [103]. Patterns are also named and catalogued, allowing architecture knowledge to be communicated and shared with minimal ambiguity. Besides the design decisions, the rationale along with the context in which the pattern is applicable are also available. As a communication tool, pattern is vital during system evolution, as mention in ***section 6.2***, common cause of erosion is the limited knowledge and knowledge loss, in this case is unavailable or misunderstood design decisions. These problems can be solved by using the architecture patterns to document the design decision.

According to Gurp et al. [59], the developers can consult the patterns to avoid taking bad design decisions. Ram et al. [104] propose a pattern oriented approach to software development using patterns as building blocks of architecture. They argued that design evolution in software can best be achieved by replacing existing patterns in the design with new patterns addressing present requirements. It means that when requirements change in a system, the corresponding design could easily be identified in the form of patterns and could be accommodated in the system by replacing an existing pattern. Similarly Harrison et al. [48] proposed to use pattern to capture the architectural decision. They claimed that architecture patterns address these documentation challenges by capturing structural and behavioral information and encouraging architects to reflect on decisions in a way that doesn't interfere with the natural architectural design process.

Examples of architecture pattern are the Model-View-Controller [105], Pipe and Filter [103], and the Blackboard model [106].

8.5.1.2 Architecture anti-pattern

An anti-pattern is a literary form describing a bad solution to a recurring design problem that has a negative impact on the quality of a system design [107]. Contrary to design patterns, anti-patterns describe what not to do, but it also provides a solution for what should be done in the same case. Anti-pattern can also be used as communication tool to avoid the same bad design decisions.

Examples of architecture anti pattern are Reinvent the Wheel, Swiss Army Knife, and Vendor Lock-In [108].

8.5.1.3 Framework

Architecture frameworks provide a set of tools to systematically guide the design, documentation, implementation and evolution of the system. Most architecture frameworks provide some core functionality that can be reused or extended to build applications with the support of accompanying tools, guidelines, testing procedures and management processes.

Examples of framework are Spring [109], Struts [110] and the Zachman Framework for Enterprise Architecture [111].

8.5.1.4 Review design and code

According to Dhami et al. [112] the system architect should not only focus on creating system, he/she must also constantly review, assess and improve existing system. To keep the prescriptive and descriptive architecture aligned and consistent, review and assessment techniques are of utmost importance. In fact, every design should be reviewed and approved by someone whose responsibilities are for long-term future of the product. The review should be carried out when the design is first proposed and long before implementation [21].

Gurp et al. [59] argued that, code reviews and Fagan inspections will help to detect problems with respect to code quality and design problems. Schrøder et al. [95] claimed in their study that the developers and architects have each different goal during review process. The developer mainly wants to implement new features, while the architect wants to check architecture conformance. One of conclusion in their study is that the review processes not only align the understanding of the system, but also help to increase the awareness of design and code quality within the team.

8.5.2 Code generation

Code generation tools normally use a specification of the prescriptive architecture to produce code stubs, template or skeleton classes. However, a manually work is needed to be done before the tool can produce useful result. This work contains the mapping the prescriptive architecture into UML model or using Dependency Constraint Language as mentioned in ***section 437.2.3.2.*** The mapping capitalizes on enhancements in UML 2.0 to model dynamic and structural aspects as architectural level abstractions that can be transformed into code. Component diagrams are suggested for modelling static structures and sequence diagrams for modelling behavior.

Examples of those tools are ArchStudio [113] and Enterprise Architect [114].

8.5.3 Refactoring

Refactoring is defined according to Mens et al [115] as: “The *process of changing* a software system in such a way that it does *not alter* the external *behavior* of the code, yet improves its internal *structure*”. Refactoring in higher levels of abstraction like architectural levels in called architectural refactoring. Refactoring is commonly applied to code, but refactoring can also be applied to other development artifacts like databases, UML models, and software architecture. Refactoring software architecture is particularly relevant because during development the architecture is constantly changing. Software architecture refactoring should happen regularly during the development cycle [116].

Architectural refactoring is used to achieve quality attributes as performance, reliability, availability or modifiability. Agilists use refactoring as their main practice to accommodate changes as they come up during software development life cycle, as well as design improvement. Moser et al. [117] claimed that refactoring will increases software understandability improves software design and accelerates the coding process and the ability to find bugs. Barow [116] argued that an iterative and consistent architectural refactoring process will not only increases everyone’s confidence in the architecture and in the system as a whole, but also gives everyone a better understanding of the architecture.

According to Barow [118] architectural refactoring makes sure wrong or inappropriate decisions can be detected and eliminated early. Architectural refactoring can simplify the system software, which is one of the principles of Agile development. Barrow concluded that for avoiding architectural erosion, we need to focus on simplicity in both the software being developed and in the development process. Whenever possible, actively work to eliminate complexity from the system.

Sharifloo et al. [119] proposed the Continuous Architectural Refactoring (CAR) as a practice for identifying architectural smells and deciding solutions to remove them. CAR is a practice that is applied in parallel with agile Iterations. However, this practice requires both every team’s modelling effort and the architect’s effort to integrate the models after an agile iteration.

The benefit of software architecture refactoring is uncovering problems earlier in the development cycle when they are cheaper and easier to fix.

8.5.4 Automatic test

Gurp et al. [38] argued for, that automatic regression testing can also be used to prevent the architectural erosion. The automatic regression testing can prevent new defects from being introduced during defect fixing. Automated tests can be used to verify that the system still works against the requirements.

Together with refactoring, automatic test can ensure the confident in the team that everything is still working after refactoring.

8.6 Documentation

As presented in the ***section 5.2***, a well-documented architecture is essential for preventing architectural erosion. However, the high effort for elaboration and maintenance of architecture documentation hinder its acceptance in practice. In the latest year there was a paradigm shift towards documenting design decisions [120]. Tyree et al [121] claimed that design decisions can be a first-class status and explicitly document and an effective tool to help stakeholders understand the architecture. These decisions can provide a concrete direction for implementation and serve as an effective tool for communication to customers and management. According to Caracciolo et al. [122] architectural erosion can also be minimized by documenting design decisions with the intent of disseminating knowledge and raising awareness.

Kruchten [123] and Burge et al. [124] proposed a proactive approach to prevent architectural erosion. The approach was focusing on using design rationales to document architectural decisions. They argued that explicitly recording design decisions, justifications, alternatives, and conflicting perspectives, is necessary in order to preserve architectural qualities.

8.7 Discussion

The common of those approaches are most of them are proactive approaches, which are perfectly suitable to prevent the architectural erosion. But as mention in ***section 8.1*** to have a chance to avoid the architectural erosion, it requires that the top management is aware and understand of the problem. The solution to the architectural erosion is not simple, thus it not only needs commitment from the top management in the organization, but also a support from all the levels within the organization. But if management support is provided to developers, they can implement different patterns to stop the erosion effects depending upon the availability of tools, domain of the project, and perhaps rising culture where fighting erosion is treasured. *Organization* and *Team/people* approaches are just a long journey, because it is not easy to change the mindset of everyone to go in the same direction.

[Confidential material removed]

The *Development process and model* approaches are useful in identifying early stages of architecture erosion, where minor violations of architectural principles take place. This allows the descriptive architecture to be kept in step with the prescriptive architecture. Repairing the violation at this stage is also a small cost. But the cost of effort to define, implement and maintain it over a product life cycle is very high. Plus the success and effectiveness of those approaches are largely dependent on human factors. Process guidelines are often ignored and process checks bypassed in favor of achieving project deadlines or cutting costs. Using tools will help the process and reduce the cost, but it will also create another dependency and complexity. Choosing the right tools and right process is big dilemma in a lot of industry companies. Forrester Research [88] has concluded in their survey report that even though the availability of number dependency and static analysis tools are large, those tools are not widely used in industry projects. Most of companies still rely on manual reviews for checking architectural conformance at code level or no use of compliance monitoring methods.

The *coding* approach *Design enforcement* depends entirely on the competence of the architects and developers to transfer the prescriptive architecture model into the architecture pattern, framework and the anti-pattern choices. If this model representative of the implementation is not accurate, then the descriptive architecture has already diverged from the prescriptive architecture. The effectiveness of those approaches will reflect on how precisely an architectural design can be transformed and how the project team understand the prescriptive architecture. Therefor documenting the decisions is an important task, which should be taken with care. Similarly, the *code generation* depends on the model representative of prescriptive architecture and the glue codes, which maps the prescriptive architecture and the descriptive architecture. The *refactoring* approach is an effective approach to for improving the design of an existing code base and with that eliminate the violations. But according to Schumacher [62] there are situation, when it is less obvious where and what kind of refactoring would improve the critical part of the system. To solve this issue, Fowler et al. [125] proposed concept of code smells to help developers identify design flaws in their software. Each code smell has a set of refactoring techniques associated with it. *Automatic test* should be a part of the Continuous Integration process, which is a quality gate for all the projects. But again, the big effort to implement and maintain the test can hinder the widespread of the approach.

Even though *documentation* is vital for long-life projects, it suffers the same fate as the *Development process and model,* because it is costly and big effort required for creating and maintains it. As Gurp [59] has pointed out that the source code is usually the documentation. Consequently, many of the concepts used during the design phase are represented in an implicit fashion. This causes serious maintenance issues since maintainers will have to reconstruct the design from the source code before they can change it.

As we go through all the approaches, we don’t see any approach, which can stand alone as a perfect solution for avoiding architectural erosion. Most of the approach is somehow related and depend on the others, therefor as we mention earlier, we think awareness is the most important factor in fighting against architectural erosion.

9. Summary

This report is written based on a systematic literature review about the architectural erosion. The output of the report is:

- A list of reasons for why architectural erosion happens.
- A list of the techniques for detecting architectural erosion.
- A list of the approaches for avoiding architectural erosion.

Besides drawing a comparison to the AK2 system during the review, we also draw a relation between the reasons of architectural erosion and the approaches for avoiding architectural erosion by classifying them in the same categories. ***Figure 19*** illustrates the summary of the report.

The result of the report shows that, one of the ways to avoid the architectural erosion is awareness. But it needs to be applied in all the levels within the organization:

- The top management needs to be aware of the architectural erosion problem. But it is not enough to know about it, they also must know what is needed to do to build a "right" culture.
- Besides knowing about the architectural erosion and how to fight it, the architects also need to communicate the architecture to the stakeholders to create a common understanding about the architecture.
- The developers need to learn the basic skill to both understand and implement the architectural patterns. They also need to know about the architectural erosion and how to fight it.

Architectural erosion is unavoidable, but we can minimize its damage by detecting it early.

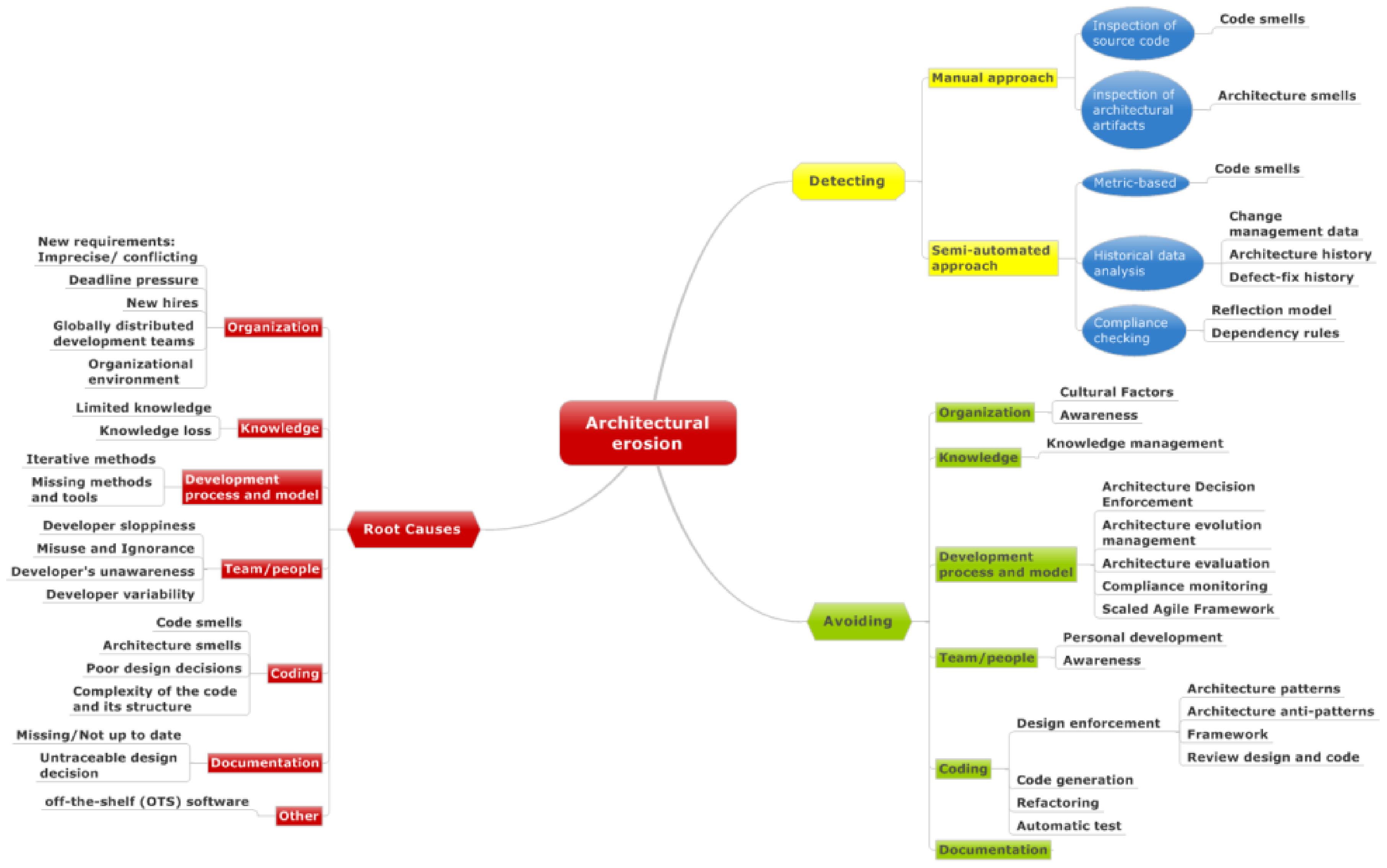


Figure 19 Report summary

10. Conclusion and Future Works

We have performed a literature review conducted on the topic of architectural erosion, using most of the techniques of a systematic literature review. A total of 108 primary studies were selected using a well-defined review protocol, which is described in ***section 4***. The summary of this report is illustrated in the ***Figure 19***. The four contributions of this report are the following:

- The conclusion of ***H1*** is presented in the ***section 6***, where we classified and reported seven categories of the reason for architectural erosion. Most of the papers pointing out *New requirements: Imprecise/ conflicting* as the main reason for architectural erosion. It is also a reason, we have found in our AK2 system case study. As discussed in ***section 8.3.1***, personal development needs to get more attention from the top management.

- A comparison to the AK2 case study to find the similarity between the found architectural erosion root causes and these occurred in the AK2 system. The result is presented in ***section 6.8.2.***

- We have also classified the techniques to detect the architectural erosion, where the compliant checking is the most used approach in all the reviewed papers. However, as discussed under ***section 7.3***, the approach is not widespread in the industry because of the big effort on the mapping between the prescriptive architecture and the descriptive architecture. The conclusion of **H3** is presented in section 7.

- In the ***section 8*** we presented a survey of the existing approach for avoiding architectural erosion, which is an answer to **H4**. The approaches as classified in the same categories as the root cause, to map the relation between the two. The result is showing that there is no one size fit all solution; one must work hard to find out the best approach that seems suitable for one's organization and projects.

My definitions of well-documented architecture changed during the study. To keep the architecture consistent we need to perform continuous comfort architecture check. But to be able to do that, then we need to map the prescriptive architecture to the implementation. These mappings allowed us to trace the influence of a code anomaly on architecture design. These mappings also allowed us to identify how the modularization of concerns in the code was related to architecture problems.

The most important lesson learnt is that architectural erosion is inevitable, it is only sensible to manage that erosion as well as possible. If we want to have a chance to eliminate the architectural violations, they need to be detected as soon as possible. According to McConnell [126] the time the team spends finding and correcting that architecture error is 50 to 200 times less than the time it would have spent detecting and correcting that same error during construction or system testing.

If you're going to get ***bad news*** *on a software project,*
it's best to get it ***early*** *because of the*
upstream/downstream effect on ***project costs****.*

Steve McConnell [126]

The second lesson learnt is that when we have a long life software system, it is very important to have focus on the technical debt in the whole software system life cycle. The team should take responsibility for reducing technical debt. Team members should discuss technical debt and give it the right priority on the product backlog.

Through the review we have recognized that even though *Organization* and *Team/people* have a big impact on driving the architectural erosion on the positive or negative direction, there are not many papers relating the *Organization* and *Team/people* to the architectural erosion. If we look at the detecting techniques, which we presented in the ***section 7***, most of them only focus on the technical aspect. But fighting architectural erosion requires that, everyone within the organization, go for the same direction. Everyone needs to take responsibility depending on their role. The role of a manager is that he/she should create a culture where fighting software erosion is encouraged and supported. If this step is not taken, then no one will care about erosion. Similarly, for the role of architect or a developer, he/she should educate himself about the different causes of erosion and the different approaches to fight it.

In the future, we would like to perform empirical studies applying the found detecting techniques and avoiding approaches on industrial cases. We also want to make a survey on the existing tool sets for both detecting and avoiding architectural erosion.

11. Appendix A: papers included in the review

[L1] M. Dalgarno. (2009, 30-08-2017). *When Good Architecture Goes Bad*. Available: http://www.methodsandtools.com/archive/archive.php?id=85

[L2] M. B. Bo Zhang, Thomas Patzke, Krzysztof Sierszecki, Juha Erik Savolainen "Variability evolution and erosion in industrial product lines: a case study ", August 2013 SPLC '13: Proceedings of the 17th International Software Product Line Conference 2013.

[L3] U. M. Ahmad Nida, Ikram, Naveed "Value-based software architecture knowledge management tool," Proceedings of the IASTED International Conference on Software Engineering, SE 2010 2010.

[L4] J. V. Medvidovic Nenad, "Using software evolution to focus architectural recovery," Automated Software Engineering 2006.

[L5] A. P. Harrison Neil B., Zdun Uwe, "Using patterns to capture architectural decisions," IEEE Software2007.

[L6] L. M. F. Caracciolo Andrea, Nierstrasz Oscar "A Unified Approach to Architecture Conformance Checking," Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015 2015.

[L7] S. J. Sebastian Gerdes, Matthias Riebisch, Sandra Schröder, Mohamed Soliman, Tilmann Stehle "Towards the essentials of architecture documentation for avoiding architecture erosion ", November 2016 ECSAW '16: Proccedings of the 10th European Conference on Software Architecture Workshops 2016.

[L8] G. B. Rainer Weinreich, "Towards supporting the software architecture life cycle," Journal of Systems and Software, v 85, n 1, January 2012 2012.

[L9] S. H. Matthias Mair, Andreas Rausch "Towards flexible automated software architecture erosion diagnosis and treatment ", April 2014 WICSA '14 Companion: Proceedings of the WICSA 2014 Companion Volume 2014.

[L10] S. H. Matthias Mair, "Towards Extensive Software Architecture Erosion Repairs," Proceeding ECSA'13 Proceedings of the 7th European conference on Software Architecture 2013.

[L11] D. O. V. M. T. Terra Ricardo, "Towards a dependency constraint language to manage software architectures," Software Architecture - Second European Conference, ECSA 2008, Proceedings 2008.

[L12] D. P. Joshua Garcia, George Edwards, Nenad Medvidovic, "Toward a Catalogue of Architectural Bad Smells," Proceeding QoSA '09 Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems 2009.

[L13] R. R. Francesca Arcelli Fontana, Marco Zanoni "Tool support for evaluating architectural debt of an existing system: an experience report ", April 2016 SAC '16: Proceedings of the 31st Annual ACM Symposium on Applied Computing 2016.

[L14] H. M. Mokni Abderrahman, Urtado Christelle, Vauttier Sylvain, Zhang Huaxi "A three-level formal model for software architecture evolution," CEUR Workshop Proceedings 2014.

[L15] P. Inverardi, "Taming the uncertainty: variability as a means for predictable system evolution," January 2013 VaMoS '13: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems 2013.

[L16] T. H. Srdjan Stevanetic, Uwe Zdun, "Supporting Software Evolution by Integrating DSL-based Architectural Abstraction and Understandability Related Metrics," Proceeding ECSAW '14 Proceedings of the 2014 European Conference on Software Architecture Workshops Article No. 19 2014.

[L17] J. L. Catherine Blake Jaktman, Ming Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study," IFIP — The

International Federation for Information Processing 1999.

[L18] B. Merkle, "Stopping (and reversing) the architectural erosion of software systems. An industrial case study," Research& Development, Software Engineering SICK AG Waldkirch, Germany 2010.

[L19] B. Merkle, "Stop the software architecture erosion: building better software systems ", October 2010 OOPSLA '10: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion 2010.

[L20] A. E. Nenad Medvidovic, "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery," ICSE'03 International Conference on Software Engineering Portland, Oregon 2003.

[L21] T. R. Passos Leonardo, Valente Marco Tulio, Diniz Renato, Das Chagas Mendonça Nabor, "Static architecture-conformance checking: An illustrative overview," IEEE Software 2010.

[L22] M. S. Bruce Anderson, Larry Best, Kent Beck "Software architecture: the next step for object technology (panel)," Proceeding OOPSLA '93 Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications 1993.

[L23] J. B. Anton Jansen, "Software Architecture as a Set of Architectural Design Decisions," Proceeding WICSA '05 Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture 2005.

[L24] D. L. Parnas, "Software Aging," Proceeding ICSE '94 Proceedings of the 16th international conference on Software engineering 1994.

[L25] C. T. Birgit Boss, Sreejith Krishnan, Arun Nutakki, Vinod Gopinath "Setting up architectural SW health builds in a new product line generation ", November 2016 ECSAW '16: Proccedings of the 10th European Conference on Software Architecture Workshops 2016.

[L26] J. B. Sebastian Herold, Andreas Rausch "Second Workshop on Software Architecture Erosion and Architectural Consistency (SAEroCon 2015) ", September 2015 ECSAW '15: Proceedings of the 2015 European Conference on Software Architecture Workshops 2015.

[L27] H. Ahn, "Reconstruction of runtime software architecture for object-oriented systems ", April 2015 SAC '15: Proceedings of the 30th Annual ACM Symposium on Applied Computing 2015.

[L28] D. K. Naim Sheikh Motahar, Hossain M. Shahriar "Reconstructing and evolving software architectures using a coordinated clustering framework," Automated Software Engineering 2017.

[L29] N. E. Haitzer Thomas, Zdun Uwe "Reconciling software architecture and source code in support of software evolution," Journal of Systems and Software, v 85, n 1, January 2012 2012.

[L30] V. M. T. Terra Ricardo, Czarnecki Krzysztof, Bigonha Roberto S. , "Recommending refactorings to reverse software architecture erosion," Proceedings - 2012 16th European Conference on Software Maintenance and Reengineering, CSMR 2012 2012.

[L31] M. T. V. Ricardo Terra, Krzysztof Czarnecki, Roberto S. Bigonha, "A recommendation system for repairing violations detected by static architecture conformance checking," Journal Software—Practice & Experience 2015.

[L32] E. G. Roberta Arcoverde, Isela Macia "Prioritization of Code Anomalies Based on Architecture Sensitiveness," IEEE 2013.

[L33] P. I. De Silva Mahesh, "Preventing software architecture erosion through static architecture conformance checking," 2015 IEEE 10th International Conference on Industrial and Information Systems, ICIIS 2015 - Conference Proceedings 2015.

[L34] M. Mirakhorli, "Preventing erosion of architectural tactics through their strategic implementation, preservation, and visualization," 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 – Proceedings 2013.

[L35] L. M. M., "On Understanding Laws. Evolution, and Conservation in the Large-Program Life Cycle," Journal of Systems and Software 1 1980.

[L36] R. A. Isela Macia, Alessandro Garcia, Christina Chavez, Arndt von Staa "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms," Proceeding CSMR '12 Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering 2012.

[L37] F. A. Humaira Farid, *M. Aqeel Iqbal, "MINIMIZING THE RISK OF ARCHITECTURAL DECAY BY USING ARCHITECTURE-CENTRIC EVOLUTION PROCESS," International Journal of Computer Science, Engineering and Applications (IJCSEA) 2011.

[L38] D. D. Koziolek Heiko, Goldschmidt Thomas, Vorst Philipp, "Measuring architecture sustainability," IEEE Software 2013.

[L39] C. B. Strasser Arthur, Gernert Christoph, Knieke Christoph, Körner Marco, Niebuhr Dirk, Peters Henrik, Rausch Andreas, Brox Oliver, Jauns-Seyfried Stefanie, Jelden Hanno, Klie Stefan, Krämer Michael "Mastering erosion of software architecture in automotive software product lines," SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Proceedings 2014.

[L40] D. B. Claire Dimech, "Maintaining Architectural Conformance during Software Development," Proceeding ECSA'13 Proceedings of the 7th European conference on Software Architecture 2013.

[L41] D. R. Martin Feilkas, Elmar Jurgens "The loss of architectural knowledge during system evolution: An industrial case study ", Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on 2009.

[L42] M. P. O'Reilly C., Bustard D., "Lightweight prevention of architectural erosion," Proceedings - 6th International Workshop on Principles of Software Evolution, IWPSE 2003, in Association with ESEC/FSE 2003 2003.

[L43] M. G. Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, Andrea de Lucia "Improving software modularization via automated analysis of latent topics and dependencies ", February 2014 ACM Transactions on Software Engineering and Methodology (TOSEM): Volume 23 Issue 1, February 2014 2014.

[L44] M. B. Thomas Patzke, Michaela Steffens, Krzysztof Sierszecki, Juha Erik Savolainen, Thomas Fogdal, "Identifying improvement potential in evolving product line infrastructures: 3 case studies," Proceeding SPLC '12 Proceedings of the 16th International Software Product Line Conference 2012.

[L45] D. P. Joshua Garcia, George Edwards, Nenad Medvidovic "Identifying Architectural Bad Smells," Proceeding CSMR '09 Proceedings of the 2009 European Conference on Software Maintenance and Reengineering 2009.

[L46] M. Stal, "Good is not good enough: evaluating and improving software architecture ", June 2011 QoSA-ISARCS '11: Proceedings of the joint ACM SIGSOFT conference -- QoSA and ACM SIGSOFT symposium 2011.

[L47] C. K. Benjamin Cool, Andreas Rausch, Mirco Schindler, Arthur Strasser, Martin Vogel, Oliver Brox,Stefanie Jauns-Seyfried, "From product architectures to a managed automotive software product line architecture," April 2016 SAC '16: Proceedings of the 31st Annual ACM Symposium on Applied Computing 2016.

[L48] S. R. Sundus Ayyaz, Usman Qamar, "A Four Method Framework for Fighting Software Architecture Erosion," International Journal of Computer, Electrical, Automation, Control and Information Engineering.2015.

[L49] M. N. Ding Lei, "Focus: A light-weight, incremental approach to software architecture recovery and evolution," Proceedings - Working IEEE/IFIP Conference on Software Architecture, WICSA 2001 2001.

[L50] L. H. Florian Deissenboeck, Benjamin Hummel, Elmar Juergens "Flexible architecture conformance assessment with ConQAT ", May 2010 ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 2010.

[L51] R. R. Fontana Francesca Arcelli, Zanoni Marco, Raibulet Claudia, Capilla Rafael "An experience report on detecting and repairing software ", Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016 2016.

[L52] N. Mokni Abderrahman (LGI2P / Ecole des Mines D'Alès, France), Huchard Marianne, Urtado Christelle, Vauttier Sylvain, Zhang Yulin, "An evolution management model for multi-level component-based software architectures," Proceedings - SEKE 2015: 27th International Conference on Software Engineering and Knowledge Engineering 2015.

[L53] M. B. Sebastian Herold, Jim Buckley, "Evidence in architecture degradation and consistency checking research: preliminary results from a literature review ", Proceeding ECSAW '16 Proccedings of the 10th European Conference on Software Architecture Workshops 2016.

[L54] D. T. Tobias Olsson, Morgan Ericsson, Anna Wingkvist "Evaluation of an architectural conformance checking software service ", November 2016 ECSAW '16: Proccedings of the 10th European Conference on Software Architecture Workshops 2016.

[L55] R. M. S. Ram D. Janaki, "Enabling design evolution in software through pattern oriented approach," Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2003.

[L56] B. J. W. Ajay Bandi, Edward B. Allen "Empirical evidence of code decay: A systematic mapping study," IEEE Reverse Engineering (WCRE), 2013 20th Working Conference on 2013.

[L57] P. C. R.T. Tvedt, M. Lindvall "Does the code match the design? A process for architecture evaluation ", Software Maintenance, 2002. Proceedings. International Conference on 2002.

[L58] T. L. G. Stephen G. Eick, Alan F. Karr, J. S. Marron, Audris Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," Journal IEEE Transactions on Software Engineering 2011.

[L59] M. E. Sebastian Herold, Jim Buckley, Steve Counsell, Mel Ó Cinnéide "Detection of Violation Causes in Reflexion Models," IEEE 2015.

[L60] Y. C. Sunny Wong, Michael Dalton, "Detecting Design Defects Caused by Design Rule Violations," Researchgate 2010.

[L61] S. Y. Zhang Lei, Song Hui, Chauvel Franck, Mei Hong "Detecting architecture erosion by design decision of architectural pattern," SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering 2011.

[L62] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," Proceeding ICSE '11 Proceedings of the 33rd International Conference on Software Engineering 2009.

[L63] S. B. Jilles van Gurp, Jan Bosch, "Design preservation over subsequent releases of a software product," Wiley InterScience 2005.

[L64] J. B. Jilles van Gurp, "Design erosion: problems and causes," Journal of Systems and Software 2002.

[L65] V. M. T. Terra Ricardo, "A dependency constraint language to manage object-oriented software architectures," Software - Practice and Experience 2009.

[L66] Y.-G. G. Naouel Moha, Laurence Duchien, Anne-Francoise Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," Journal IEEE Transactions on Software Engineering2010.

[L67] B. D. De Silva Lakshitha, "Controlling software architecture erosion: A survey," Journal of Systems and Software, v 85, n 1, January 2012 2012.

[L68] A. R. Sebastian Herold, "Complementing model-driven development for the detection of software architecture erosion ", May 2013 MiSE '13: Proceedings of the 5th International Workshop on Modeling in Software Engineering 2013.

[L69] C. D. A. C. Stringfellow, D. Potnuri, A. Andrews, M. Georg, "Comparison of software architecture reverse engineering methods," Information and

Software Technology 2006.
[L70] M. L. Lorin Hochstein, "Combating architectural degeneration a survey," Journal Information and Software Technology 2005.
[L71] M. Langhammer, "Co-evolution of component-based architecture-model and object-oriented source code ", June 2013 WCOP '13: Proceedings of the 18th international doctoral symposium on Components and architecture 2013.
[L72] M. M. Herold Sebastian, Rausch Andreas, Schindler Ingrid "Checking conformance with reference architectures: A case study," Proceedings - 17th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2013 2013.
[L73] J. C. C. Byron J. Williams, "Characterizing software architecture changes: A systematic review," Journal Information and Software Technology 2010.
[L74] P. N. R. Mathieu Lavallée, "Causes of premature aging during software development: An observational study," Proceedings of the 12th International Workshop on Principles on Software Evolution 2011.
[L75] D. M. Mikael Lindvall, "Bridging the Software Architecture Gap," Journal Computer2008.
[L76] I. M. Alessandro Gurgel, Alessandro Garcia, Arndt von Staa, Mira Mezini, Michael Eichberg, Ralf Mitschke "Blending and reusing rules for architectural degradation prevention ", April 2014 MODULARITY '14: Proceedings of the 13th international conference on Modularity 2014.
[L77] R. T. M. Lindvall, P. Costa "Avoiding architectural degeneration: an evaluation process for software architecture ", Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on 2002.
[L78] U. C. Zhang Huaxi, Vauttier Sylvain, "Architecture-centric development and evolution processes for component-based software," Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering 2010.
[L79] R. M. Schröder Sandra, Soliman Mohamed, "Architecture enforcement concerns and activities - An expert study," Software Architecture - 10th European Conference, ECSA 2016, Proceedings 2016.
[L80] E. R. J. Sergio Miranda, Marco Tulio Valente, Ricardo Terr, "Architecture Conformance Checking in Dynamically Typed Languages," AITO — Association Internationale pour les Technologies Objets 2016.
[L81] De Silva, L. B., Dharini (2012). Controlling software architecture erosion: A survey. Retrieved from Journal of Systems and Software, v 85, n 1, January 2012
[L82] Muhammad S. Mehwish Riaz, Husnain Naqvi, "Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture," International Conference on Advanced Software Engineering and Its Applications 2009.
[L83] J. G. Isela Macia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, Arndt von Staa "Architectural Consistency Checking in Plugin-Based Software Systems ", Proceedings of the 2015 European Conference on Software Architecture Workshops 2015.
[L84] M. K. Greifenberg Timo, Rumpe Bernhard, "Architectural consistency checking in plugin-based software systems," Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW 2015 2015.
[L85] N. E. Haitzer Thomas, Zdun Uwe, "Architecting for decision making about code evolution," Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW 20152015.
[L86] D. Dike!, Kane, D., Ornburn, S., Loftus, B., and Wilson, J., "Applying Software Product-Line Architecture," IEEE Software. 1997.
[L87] A. K. H. P. S. Dham, "Analysis of Software Design Erosion Issue," International Journal of Advanced Research in Computer Science and Software Engineeri 2013.
[L88] S. Barow, "Architecture Erosion in Agile Development," [Online]. Available:

http://lattix.com/blog/2017/03/31/architecture-erosion-agile-development. [Accessed 17 06 2017].

[L89] G. H. El-Khawaga, Galal Hassan Galal-Edeen, A.M. Riad , " ARCHITECTING IN THECONTEXT OF AGILE SOFTWARE DEVELOPMENT: FRAGILITY VERSUS FLEXIBILITY "International Journal of Computer Science, Engineering and Applications (IJCSEA) Vol.3 2013.

[L90] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in ESEM, 2010

[L91] J. Rosik, A. L. Gear, J. Buckley, M. A. Babar, and D. Connolly, "Assessing architectural drift in commercial software development: A case study," Software–Practice and Experience, vol. 41, 2011

[L92] S. M. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in 3rd ESEM, 2009

[L93] T. Olsson, M. Ericsson and A. Wingkvist, "Motivation and Impact of Modeling Erosion using Static Architecture Conformance Checking", IEEE International Conference on Software Architecture Workshops, 2017.

[L94] E. Bouwers and A. van Deursen, "A Lightweight Sanity Check for Implemented Architectures", IEEE Software, 2010.

[L95] M. Lanza and R. Marinescu, Object-oriented metrics in practice. Berlin: Springer, 2011.

[L96] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws", IEEE, 2004.

[L97] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in Java source-code", IEEE, 2005.

[L98] S. Herold and J. Buckley, "Feature-Oriented Reflexion Modelling", Proceeding ECSAW '15 Proceedings of the 2015 European Conference, 2015.

[L99] R. Koschke, "Incremental Reflexion Analysis", IEEE, 2010.

[L100] J. Buckley, N. Ali, M. English, J. Rosik and S. Herold, "Real-Time Reflexion Modelling in Architecture Reconciliation", Journal Information and Software Technology Volume 61, 2015.

[L101] R. Koschke and D. Simon, "Hierarchical reflexion models", IEEE, 2003.

[L102] R. Terra and M. Tulio Valente, "A dependency constraint language to manage object-oriented software architectures", Journal Software—Practice & Experience Volume 39, 2009.

[L103] L. Pruijt, C. Koppe and S. Brinkkemper, "Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparative Study of Tool Support", IEEE, 2013.

[L104] N. Sangal, E. Jordan, V. Sinha and D. Jackson, "Using dependency models to manage complex software architecture", Proceedings of the 20th Annual ACM SIGPLAN Conference, 2005.

[L105] J. Brunet, R. A. Bittencourt and J. F. Dalton Serey, "On the Evolutionary Nature of Architectural Violations," IEEE , 2012.

[L106] Z. Li and J. Long, "A Case Study of Measuring Degeneration of Software Architectures from a Defect Perspective," IEEE, 2011.

[L107] M. C. Ohlsson, A. v. Mayrhauser, B. McGuire and C. Wohlin, "Code decay analysis of legacy software through successive releases," IEEE Arerospace Conference, 1999.

[L108] A. V. Mayrhauser, J. Wang, M. Ohlsson and C. Wohlin, "Deriving a fault architecture from defect history," IEEE Computer Society Press, 1999.

Reference

[1] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Software_evolution. [Accessed 27- 08 2017].

[2] B. Kitchenman, "Procedures for Performing Systematic Reviews," ABB Corporate Research, 2004.

[3] "The Software Engineering Institute," [Online]. Available: http://www.sei.cmu.edu/architecture/definitions.html. [Accessed 27 05 2017].

[4] L. Bass, Clements P and R. Karzman , Software Architecture in Pratice., Addison-Wesley, 2. edition, 2003.

[5] M. James and et al., A Practical Guide to Enterprise Architecture, Prentice Hall, 2004.

[6] I. C. Society, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," IEEE Std 1472000, 2000.

[7] P. Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley Professional, Third Edition. 2003.

[8] O. M. G. Inc., OMG Unified Modeling Language Specification Version 1.5,, Document number 03-03-01. March 2003.

[9] S. Brown, Software Architecture for Developers, Lean Publishing, 2014.

[10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2002.

[11] "ISO/IEC/IEEE 42010:2011 Systems and software engineering -- Architecture description," International Organization for Standardization, 2011.

[12] R. N. D. S. Christine Hofmeister, Applied Software Architecture, Addison-Wesley, 2000.

[13] H. B. Christensen, A. Corry and K. M. Hansen, The 3+1 Approach to Software Architecture Description Using UML, Department of Computer Science, University of Aarhus Revision 2.3, 2012.

[14] H. C. Perla Velasco-Elizondo, "On Software Architecture Processes and their Use in Practice," IGI Global, 2014.

[15] M. R. Barbacci, R. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock and W. G. Wood, "Quality Attribute Workshops (QAWs), Third Edition," Software Engineering Institute, August 2003.

[16] A. J. Lattanze, "Architecting Software Intensive Systems: A Practitioners Guide," Boca Raton, FL: CRC. Press, 2009.

[17] R. Kazman, M. Klein and P. Clements, "ATAM: Method for Architecture Evaluation," Software Engineering Institute, 2000.

[18] H. B. Christensen, K. M. Hansen and K. R. Schougaard, "An Empirical Study of Software Architects' Concerns," Asia-Pacific Software Engineering Conference, IEEE Computer Society, 2009.

[19] P. Kruchten, "Architectural Blueprints—The 4+1 View Model of Software Architecture," IEEE, 1995.

[20] A. Hassan and R. Holt, "Using development history sticky notes to understand software architecture," Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on , 2004.

[21] D. L. Parnas, " Software aging," Proceedings of ICSE 1994, 1994.

[22] J. Highsmith, "Agile Project Management: Creating Innovative Products," Pearson Education/Addison-Wesley, 2004.

[23] R. Fjeldstad and W. Hamlen, "Application program maintenance-report to to our respondents," IEEE Computer Soc. Press., 1983.

[24] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," ACM Software, 1992.

[25] M. T. V. K. C. R. S. B. Ricardo Terra, "Recommending Refactorings to Reverse Software Architecture Erosion," 16th European Conference on Software Maintenance and Reengineering, 2012.

[26] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel and G. Antoniol, "ADvISE: Architectural Decay In Software Evolution," Software Maintenance and Reengineering (CSMR), 2012 16th European Conference, 2012.

[27] L. Hochstein and M. Lindvall, "Combating architectural degeneration: a survey," Information and Software Technology 47, 2005.

[28] M. Dalgarno, "When Good Architecture Goes Bad," 2009. [Online]. Available: http://www.methodsandtools.com/archive/archive.php?id=85.

[29] J. v. Gurp and J. Bosch, "Design erosion: Problems and causes," Journal of Systems and Software 61, 2002.

[30] C. Izurieta and J. M. Bieman, "How Software Designs Decay: A Pilot Study of Pattern Evolution," Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement, IEEE, 2007.

[31] R. N. Taylor, N. Medvidovic and E. Dashofy, Software Architecture :Foundations, Theory, and Practice., Wiley, 2009.

[32] M. Feilkas, D. Ratiu and E. Jurgens, "The Loss of Architectural Knowledge during System Evolution: An Industrial Case Study," IEEE 17th International Conference, 2009.

[33] M. Dalgarno, "When Good Architecture Goes Bad," [Online]. Available: http://www.methodsandtools.com/archive/archive.php?id=85. [Accessed 18 06 2017].

[34] J. B. Jilles van Gurp, "Design erosion: Problems and causes," Journal of Systems and Software, vol. 61, no. 2, 2002.

[35] L. Erlikh, "Leveraging Legacy System Dollars for E-Business," IEEE IT Pro, 2000.

[36] W. Cunningham, "The WyCash portfolio management system," Proceeding OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), 1993.

[37] D. Winkler and S. B. Bergsmann, "Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies," Springer, 2017.

[38] J. v. Gurp, S. Brinkkemper and J. Bosch, "Design preservation over subsequent releases of a software product: A case study of Baan ERP," Wiley InterScience , 2005.

[39] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2001.

[40] S. Ayyaz, S. Rehman and U. Qamar, "A Four Method Framework for Fighting Software Architecture Erosion," International Journal of Computer, Electrical, Automation, Control and Information Engineering, 2015.

[41] R. T. Tvedt, P. Costa and M. Lindvall, "Does the Code Match the Design? A Process for Architecture Evaluation," IEEE , 2003.

[42] S. Herold, M. Mair, A. Rausch and I. Schindler, "Checking Conformance with Reference Architectures: A Case Study," IEEE International Enterprise Distributed Object Computing Conference, 2013.

[43] M. Lindvall and D. Muthig, "Bridging the Software Architecture Gap," Journal Computer , 2008.

[44] M. Hugo, "Infoq," [Online]. Available: https://www.infoq.com/articles/top5-problems-distributed. [Accessed 8 10 2017].

[45] I. Macia, R. Arcoverde, A. Garcia, C. Chavez and A. v. Staa, "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms," IEEE , 2013.

[46] L. Zhang, Y. Sun, H. Song, F. Chauvel and H. Mei, "Detecting Architecture Erosion by Design Decision of Architectural Pattern," Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering, 2011.

[47] J. Bosch, "Software Architecture: The Next Step," Springer, 2004.

[48] N. B. Harrison, P. Avgeriou and U. Zdun, "Using Patterns to Capture Architectural Decisions," IEEE, 2007.

[49] "Agile Manifesto," [Online]. Available: http://agilemanifesto.org/iso/en/manifesto.html. [Accessed 02 08 2017].

[50] M. Mirakhorli, "Preserving the Quality of Architectural Tactics in Source Code," College of Computing and Digital Media Dissertations, 2014.

[51] H. P. S. Dham and A. Kumar, "Analysis of Software Design Erosion Issues," International Journal of Advanced Research in, 2013.

[52] S. Miranda, E. R. Jr, M. T. Valente and R. Terr, "Architecture Conformance Checking in Dynamically Typed Languages," AITO — Association Internationale pour les Technologies Objets, 2016.

[53] M. B. M. S. K. S. J. E. S. T. F. Thomas Patzke, "Identifying improvement potential in evolving product line infrastructures: 3 case studies," Proceeding SPLC '12 Proceedings of the 16th International Software Product Line Conference, 2012.

[54] D. B. Claire Dimech, "Maintaining Architectural Conformance during Software Development," Proceeding ECSA'13 Proceedings of the 7th European conference on Software Architecture, 2013.

[55] L. Ding and N. Medvidovic, "Focus: A Light-Weight, Incremental Approach to Software Architecture R-ecovery and Evolution," IEEE, 2002.

[56] N. Medvidovic and V. Jakobac, "Using software evolution to focus architectural recovery," Springer, 2006.

[57] N. Medvidovic, A. Egyed and P. Gruenbacher, "Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery," ICSE International Conference on Software Engineering, 2003.

[58] M. Lavallée and P. N. Robillard, "Causes of Premature Aging during Software Development: An Observational Study," ACM Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, 2011.

[59] J. v. Gurp and J. Bosch, "Design erosion: problems and causes," Journal of Systems and Software, 2002.

[60] R. Arcoverde, E. Guimaraes, I. Macia, A. Garcia and Y. Cai, "Prioritization of Code Anomalies based on Architecture Sensitiveness," IEEE, 2014.

[61] M. V. Mäntylä, J. Vanhanen and C. Lassenius, "Bad smells–humans as code critics," Proceedings. 20th IEEE International Conference, 2004.

[62] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and M. Shaw, "Building empirical support for automated code smell detection," Proceedings of the 2010 ACM-IEEE International Symposium , 2010.

[63] B. Merkle, "Stop the Software Architecture Erosion: Building better software systems," Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion , 2010.

[64] R. Martin, Agile Software Development, Principles, Patterns, and Practices, Pearson Higher Education; International ed edition , 2013.

[65] D. M. Le, C. Carrillo and R. Capilla, "Relating Architectural Decay and Sustainability of Software Systems," IEEE/IFIP Conference on Software Architecture, 2016.

[66] C. B. Jaktman, J. Leaney and a. M. Liu, "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study," Software Architecture. IFIP — The International Federation for Information Processing, vol 12, 1999.

[67] S. Gerdes, S. Jasser, M. Riebisch, S. Schröder, M. Soliman and T. Stehle, "Towards the Essentials of Architecture Documentation for Avoiding Architecture Erosion," Proceeding ECSAW '16 Proccedings of the 10th European Conference on Software Architecture Workshops , 2016.

[68] M. D. Silva and I. Perera, "Preventing Software Architecture Erosion Through Static Architecture Conformance Checking," IEEE 10th International Conference on Industrial and Information Systems, 2015.

[69] M. Software, "McCabe," [Online]. Available: http://www.mccabe.com/iq_research_metrics.htm. [Accessed 09 09 2017].

[70] J. Ganssle, "The Ganssle Group," [Online]. Available: http://www.ganssle.com/articles/cyclomaticcomplexity.html. [Accessed 09 09 2017].

[71] G. Travassos, F. Shull, M. Fredericks and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," ACM SIGPLAN Notices Issue 10 , 1999.

[72] E. Bouwers and A. v. Deursen, "A Lightweight Sanity Check for Implemented Architectures," IEEE Software, 2010.

[73] M. Lanza and M. Radu, Object-Oriented Metrics in Practice, Springer,

2006.

[74] R. Marinescu, "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws," Proceedings. 20th IEEE International Conference, 2004.

[75] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in Java source-code," IEEE Computer Society, 2005.

[76] C. Stringfellow, C. D. Amory, D. Potnuri, A. A. Andrews and M. Georg, "Comparison of software architecture reverse engineering methods".

[77] J. Brunet, R. A. Bittencourt and J. F. Dalton Serey, "On the Evolutionary Nature of Architectural Violations," IEEE , 2012.

[78] Z. Li and J. Long, "A Case Study of Measuring Degeneration of Software Architectures from a Defect Perspective," IEEE, 2011.

[79] M. C. Ohlsson, A. v. Mayrhauser, B. McGuire and C. Wohlin, "Code decay analysis of legacy software through successive releases," IEEE Arerospace Conference, 1999.

[80] A. V. Mayrhauser, J. Wang, M. Ohlsson and C. Wohlin, "Deriving a fault architecture from defect history," IEEE Computer Society Press, 1999.

[81] Gail C. Murphy, D. Notkin and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," IEEE, 2001.

[82] R. Koschke and D. Simon, "Hierarchical reflexion models," IEEE, 2003.

[83] S. Herold and J. Buckley, "Feature-Oriented Reflexion Modelling," Proceeding ECSAW '15 Proceedings of the 2015 European, 2015.

[84] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," Software Practice & Experience, 2009.

[85] L. Pruijt, C. Koppe and S. Brinkkemper, "Architecture compliance checking of semantically rich modular architectures: A comparative study of tool support," IEEE, 2013.

[86] N. Sangal, E. Jordan, V. Sinha and D. Jackson, "Using dependency models to manage complex software architecture," 20th Conference on Object-Oriented Programmingming, Systems, Languages, and Applications (OOPSLA), 2005.

[87] R. Koschke, "Incremental Reflexion Analysis," 14th European Conference on Software Maintenance and Reengineering, 2010.

[88] Forrester, "The value and importance of code reviews," Klocwork, 2010.

[89] F. Bourquin and R. K. Keller, "High-impact Refactoring Based on Architecture Violations," 11th European Conference on Software, 2007.

[90] M. Mair, "Mairma," [Online]. Available: http://www.mairma.de/archerosion.shtml. [Accessed 17 09 2017].

[91] M. Lindvall, R. Tesoriero and P. Costa, "Avoiding architectural degeneration: an evaluation process for software architecture," IEEE, 2002.

[92] T. Gilb, "Designing Maintainability in Software Engineering: a Quantified Approach," INCOSE , 2008.

[93] M. A. Babar, R. C. d. Boer, T. Dingsoyr and R. Farenhorst, "Architectural Knowlege Management Strategies: Approaches in Research and Industry," IEEE Computer Society, 2007.

[94] C. O'Reilly, P. Morrow and D. Busta, "Lightweight Prevention of Architectural Erosion," IEEE, 2003.

[95] S. Schröder, M. Riebisch and M. Soliman, "Architecture Enforcement Concerns and Activities - An Expert Study," Springer, 2016.

[96] G. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach, CRC Press, 2010.

[97] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann and N. Schuster, Reusable architectural decision models for enterprise application development, Springer, 2007.

[98] A. Jansen, J. V. D. Ven, P. Avgeriou and D. K. Hammer, "Tool Support for Architectural Decisions," IEEE, 2007.

[99] B. Westfechtel and R. Conradi, "Software architecture and software configuration management," Springer, 2003.

[100] R. Kazman, G. Abowd, L. Bass and P. Clements, "Scenario-based analysis of software architecture," IEEE Software, 1996.

[101] J. C. Dueñas, W. L. d. Oliveira and J. A. d. l. Puente, "A Software Architecture Evaluation Model," Springer, 1998.

[102] Scaled Agile, Inc, "Scaled Agile Framework," [Online]. Available: http://www.scaledagileframework.com/architectural-runway/. [Accessed 9 10 2017].

[103] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture - Volume 1: A System of Patterns, Wiley Publishing , 1996.

[104] J. Ram and Rajasree, "Enabling Design Evolution in Software through Pattern Oriented Approach," Springer, 2003.

[105] Microsoft, "MSDN," [Online]. Available: https://msdn.microsoft.com/en-us/library/ff649643.aspx. [Accessed 24 09 2017].

[106] P. Nii, "The blackboard model of problem solving," Journal AI Magazine , 1986.

[107] W. J. Brown, R. C. Malveau, H. W. ". McCormick and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, wiley, 1998.

[108] Sourcemaking, "Sourcemaking," [Online]. Available: https://sourcemaking.com/antipatterns/software-architecture-antipatterns. [Accessed 24 09 2017].

[109] "Spring," [Online]. Available: http://spring.io/. [Accessed 24 09 2017].

[110] Apache, "Struts," [Online]. Available: http://struts.apache.org/. [Accessed 24 09 2017].

[111] J. A. Zachman, "A framework for information systems architecture," IBM, 1987.

[112] H. P. S. Dhami and A. Kumar, Analysis of Software Design Erosion Issues, Ijarcsse, 2013.

[113] University of California, "Archstudio," [Online]. Available: http://isr.uci.edu/projects/archstudio/. [Accessed 24 09 2017].

[114] Sparxsystems, "Enterprise Architect," [Online]. Available:

http://sparxsystems.com/products/ea/. [Accessed 24 09 2017].

[115] T. Mens and T. Tourwé, "A survey of software refactoring," IEEE, 2004.

[116] S. Barow, "Lattix," 2017. [Online]. Available: http://lattix.com/blog/2017/07/14/motivation-software-architecture-refactoring. [Accessed 24 09 2017].

[117] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti and G. Succi, "A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team," Springer, 2008.

[118] S. Barow, "Architecture Erosion in Agile Development," [Online]. Available: http://lattix.com/blog/2017/03/31/architecture-erosion-agile-development. [Accessed 17 06 2017].

[119] A. A. Sharifloo, A. S. Saffarian and F. Shams, "Embedding Architectural Practices into Extreme Programming," IEEE, 2008.

[120] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," IEEE, 2005.

[121] J. Tyree and A. Akerman, "Architecture decisions:Demystifying architecture," IEEE, 2005.

[122] A. Caracciolo, M. F. Lungu and O. Nierstrasz, "A Unified Approach to Architecture Conformance Checking," IEEE, 2015.

[123] P. Kruchten, "An Ontology of Architectural Design Decisions in Software-Intensive Systems," Researchgate, 2004.

[124] J. E. Burge and D. C. B. b, "Software Engineering Using RATionale," Journal of Systems and Software Volume 81, 2008.

[125] M. Fowle, K. Bec, J. Brant, W. Opdyke and D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[126] S. McConnell, Software Project Survival Guide, Microsoft Press, 1997.