Architecture Enforcement Concerns and Activities - An Expert Study

Sandra Schröder, Mohamed Soliman, Matthias Riebisch

Universität Hamburg, Department Informatics, Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany

Abstract

Architecture enforcement is concerned with the correct and seamless implementation of architecture design decisions in order to ensure software quality. In a previous study, we conducted an empirical study in order to gain insight into the industrial practice of architecture enforcement. There, we asked 12 software architects from industry about their experience with the architecture enforcement process. As a result, we identified architecture enforcement concerns and activities. In this paper, we extend our contributions of the existing study. Firstly, we conducted five additional interviews with software architects from two different domains, namely the enterprise application and the automotive domain. This adds new architecture concerns and activities to the existing list. Secondly, we conducted a literature review. We compared our findings from the interviews with the results from the literature review and evaluated how architects' enforcement concerns and activities are discussed in literature. We found that several concerns and activities are already known from literature, but are not viewed in the context of architecture enforcement by the software architecture community. Lastly, we connected the discovered architecture concerns and activities with each other. Those relationships determine the reason why a specific architecture enforcement activity is conducted.

Keywords: software architecture, architecture enforcement, software architecture in industry, architecture erosion, empirical study

1. Introduction

A system's software architecture describes the principal design decisions for a software system. These decisions have the highest impact on the system's quality, and they are hard to change after their implementation (Bass et al., 2012). For example, structuring the components of the system using an architectural pattern (e.g. layers), and deciding on suitable technologies are architectural design decisions. These decisions are usually taken by an experienced software engineer, who might play the "software architect" role in a project (McBride, 2007). For example, the Rational Unified Process (RUP) (Kruchten, 2000) specifies a dedicated role for a software architect. In agile processes (e.g. Scrum (Schwaber and Beedle, 2001)), architects do not have a dedicated role, and could be involved in doing other tasks as well (Babar, 2009) (Fowler, 2003; Kruchten, 2008).

The role of a software architect is not only limited to taking design decisions (Jansen and Bosch, 2005), but it also involves many other duties (e.g. contribute to business goals of the company) (Bass et al., 2012). *Architecture enforcement* is one of the challenging architectural duties, which is concerned with the correct and seamless implementation of architectural design decisions (Zimmermann, 2009). Architecture enforcement involves two main goals:

1. *Ensure the agreement on design decisions with stakeholders*: A software architect needs to share his design decisions with stakeholders (e.g. developers), and make sure that they accept the decisions before confirming them, and starting with the implementation. The importance of this agreement is to minimize problems during the implementation of design decisions. For example, if developers are not familiar with certain patterns or technologies, they might not be able to realize design decisions. To achieve this goal, an architect needs to describe a road-map for the implementation of a system, which include fundamental guidelines that developers should follow during implementation.
2. *Check the conformance between implementation and design decisions*: An architect needs to ensure the correct implementation of design decisions during software development life cycle (i.e. during development and maintenance phases). She needs to ensure that developers follow the implementation road-map and guidelines

(Bass et al., 2012) for the implementation of design decisions, because developers could misunderstand design decisions or incorrectly implement them in so-called *architectural violations*. The accumulation of architectural violations could lead to *architecture erosion* (Perry and Wolf, 1992; Taylor et al., 2009), which results in a gradual regression for the quality of a software system (e.g. lower maintainability). To prevent architecture erosion, architects need to continuously check the adherence of implementation to architecture.

Achieving architecture enforcement faces several problems. To achieve the first goal, an architect needs to convince all stakeholders about the design decisions. This agreement process is complex and requires good communication and negotiation skills, because different stakeholders have different concerns (e.g. performance, usability, delivery time). For example, the disagreement between technical team members about technologies is a common problem. Moreover, with the growing size of software systems, it becomes time consuming to analyze source code for the adherence to architecture (achieving the second goal of architecture enforcement). To overcome these problems, methods and tools are needed to support an architect to achieve the goals of architecture enforcement.

Researchers proposed approaches to monitor and control architecture erosion e.g. Terra and Valente (2009), Pruijt et al. (2014) and Herold et al. (2013). However, current approaches did not empirically investigate architecture enforcement in practice, and especially how architects face problems for achieving architecture enforcement. This is important to ensure the usefulness of the proposed approaches and tools. Therefore, our objectives for this paper is to understand architecture enforcement in practice from the software architects' perspective, and to determine possible methods to support architecture enforcement. In details, we investigate the following research questions:

- **RQ1: What are the concerns architects consider during architecture enforcement?**

  We ask this question to determinate concrete objectives (e.g. adherence to architectural principals or properties) to achieve the goals of architecture enforcement (agreement with stakeholders on decisions and ensuring conformance between architecture and implementation). We call these objectives *Enforcement concerns*. By capturing, prioritizing and categorizing enforcement concerns from real examples in practice, we could specify empirically grounded requirements for tools to support architecture enforcement. This will additionally help providing further research directions to support architecture enforcement.

- **RQ2: What are the activities performed by architects during architecture enforcement?**

  The motivation of RQ2 is to determine current activities that an architect performs during architecture enforcement. We call these activities *enforcement activities*. By determining and understanding current enforcement activities, we determine types of scenarios, which can support developing suitable approaches for activities currently performed in practice. This will increase the probability of acceptance and usefulness for tools, because they will support practitioners in their current activities.

- **RQ3: How do architecture enforcement activities support fulfilling architecture enforcement concerns?**

  The motivation behind RQ3 is to determine the reason behind conducting each of the enforcement activities. In this way, we relate both architecture enforcement activities and concerns. Concrete relationships between architecture enforcement activities and concerns determine concrete scenarios for architecture enforcement.

To answer the research questions, we conducted first interviews with 17 experienced practitioners. The interviewed practitioners are located in different countries and work in different domains. We analyzed the transcripts of the interviews using qualitative content analysis to answer the research questions based on empirically grounded evidences from practitioners. Based on the results of the interviews, we performed a literature review to complement and refine our empirical results. In this way, we evaluate how architects' practices are actually discussed in state-of-the-art literature, and how the concepts from literature are used in practice.

The **contributions** of this paper are the following:

- We identified empirically grounded concerns and activities to support architecture enforcement. The list of identified concerns and activities extends and refines results from our previous publication (Schröder et al., 2016) to cover additional domains (e.g. Automotive, Enterprise systems).

- We determined relationships between architecture enforcement concerns and activities. The relationships provide concrete scenarios, which architects perform to achieve certain enforcement concerns.

- We identified existing approaches in literature, which recognize and analyze architecture enforcement concerns and activities, as well as architecture enforcement concerns and activities, which require additional research efforts.

The paper is structured as follows: In Section 2 we give an overview about the literature related to architecture erosion and the software architect's role. Next, in Section 3 we describe in more detail how we designed our study. In Section 6 the discovered enforcement concerns are presented. Section 7 presents the enforcement activities. Both sections include the comparison between the discovered concepts and the literature. Section 8 shows how enforcement concerns and activities are connected with each other. The results are discussed in Section 9 including a discussion about threats to validity. Section 10 concludes the paper and discusses future work.

2. Background on Architecture Enforcement

As described in Section 1, architecture enforcement aims to achieve two main goals, namely to ensure the agreement on design decisions with stakeholders and to check the conformance between implementation and design decisions. In the following, we examine and describe methods that are related with those goals.

2.1. Approaches for Ensuring the Agreement on Design Decisions

In order to ensure agreement on design decisions in the development team, it is necessary to reach a consensus about the software architecture. This means that developers should accept and be willing to implement the software architecture. Agile processes and evolutionary architectures are two methodologies prefering this viewpoint. Rather than considering architecture enforcement as a dictating process, each team member is allowed to participate in architecture design and to suggest changes if necessary. As it will be described later in this paper (Section 7), we also found that some participants favor this kind of process instead of strict and constraining enforcement activities. We will describe the role of architecture enforcement in agile processes in more detail and especially the evolutionary point of view in this section.

Agile processes favor human-centric activities like face-to-face communication and practices as daily stand up meetings or pair programming in order to enhance the shared understanding within the development team about the software system (Beck, 2000). Recently, there have been a plenty of discussions about the role of software architecture (and the software architect in particular) in agile project settings, as in (Abrahamsson et al., 2010). In order to achieve agreement on design decisions, agile processes reflect in two characteristics:

- **Agile projects avoid restrictive ivory tower architectures.** This means that instead of forcing an architecture and corresponding restrictive rules upon the development team, actively involving developers into architecting responsibilities is favored. This aims to increase the developer's understanding and acceptance of the architecture. Additionally, it increases the chance that developers are willing and have the courage to change the architecture, since they participated in the design of the architecture.

- **Software architecture is rather seen as the team's responsibility.** It is not the responsibility of a single person to design the architecture in isolation. In some project settings, however, there may exist a so-called architecture owner (Abrahamsson et al., 2010). This role has the responsibility to design the architecture upfront only to some extent, so that the development gets started in the right direction. An architecture owner collaboratively works with the team to develop and evolve the architecture. The architecture owner is mostly a technically experienced developer in the team, having a good understanding of the business domain. He may validate the designed architecture using architectural prototypes. He monitors and controls the evolution of the software architecture, but without forcing the developers to follow a strict implementation plan. They rather facilitate the creation of software architecture and mentor the development team.

Evolutionary architecture (Ford et al., 2017) helps to achieve a common mindset about software architecture in the development team and consequently to increase the agreement on architecture decisions. Instead of prescribing changes to the code, the architecture remains open for change and aims to support software architects in preserving and protecting architecture quality by enabling architectures to be continuously evolved in a changing environment.

This also allows developers to continuously give feedback about the current architecture and to propose necessary changes. In an evolutionary architecture mindset, it is assumed that an optimal architecture that meets all the requirements and constraints cannot be designed a priori. This means that architecture should be built for the current needs and should be evolved incrementally when requirements change. Following this approach, architectural problems can be detected early during development. Therefore, it reduces the need to make fixes afterwards that are potentially time-consuming and error-prone. Consequently, this can minimize architecture erosion.

Zimmermann et al. (2007) proposes an approach to support architecture decision enforcement. An important aspect of architecture decision enforcement is "sharing the results of the decision making with the stakeholders and the project team, and getting them accepted". Zimmermann et al. (2007) emphasize that it is a human-centric process mostly conducted only by manual activities like architectural templates, coaching, and code reviews. In order to support this manual work they propose to use an approach based on model transformations, called decision injection. In this approach key architectural decisions are codified as model and code transformations.This ensures that design models and code are always consistent with architectural decisions.

2.2. Approaches for checking the conformance

Architecture erosion is generally considered as the result of violating or deviating from the software architecture, as defined by (Perry and Wolf, 1992). Architecture conformance checking is a common technique in order to detect those violations and to evaluate the degree of architecture erosion. Usually, this is performed by comparing the architecture model of the intended architecture with the architecture model that was extracted from the source code. This process takes an architecture model and the source code implementing the architecture as input. Then, relevant facts from the source code are extracted and mapped to the elements of the architecture model. This is necessary in order to raise the abstraction level of the source code to the architectural level. This model of the implemented architecture can then be compared with the intended architecture model and deviations between those views can be identified. The main idea originates from the reflexion modeling approach proposed by (Murphy and Notkin, 1997). In this approach, the comparison results in a reflexion model revealing the differences between the two models by classifying the dependencies in three distinct classes, namely convergence (dependency that is allowed according to the architecture), divergence (dependency that is not allowed), and absence (dependency that was intended, but not implemented in the code).

Other approaches explicitly define architectural rules, mostly using dependency rules in order to identify violations regarding the static module view of the software architecture. Dependency rule violations can be severe since uncontrolled dependencies can increase maintenance costs, compromise independent module deployment, extensibility and testability. For example, Terra and Valente (2009) propose a domain-specific language allowing to define dependency rules that restrict how modules are allowed to access each other. Pruijt et al. (2014) define module types such as layers, components, and subsystems and relate them with specific rule types. The intended architecture can then be defined using those predefined module types. Java classes are mapped to the elements of the architecture and based on the information about the module type at hand, it is evaluated if the source code entities violate the module rule types. Herold et al. (2013) proposes a more formal approach based on first-order logic in order to allow a more flexible way to define architectural rules. Several commercial tools exist allowing static architecture conformance checking, such as Sonargraph, Structure101 or Lattix, that can support the software architect in his enforcement activities.

3. General Overview on the Research Process and Study Design

In our study we followed a qualitative research approach. We applied a process with two main phases: Practitioners Interviews and Literature Categories' Integration. For the first phase, we performed expert interviews based on a semi-structured interview guide. The interviews were transcribed and analyzed by adopting coding procedures from grounded theory, e.g. as described by (Strauss et al., 1990). As a result of this analysis, we obtain codes and categories of architecture enforcement concerns and activities. In the second phase, we examine the literature in order to find evidence about architecture enforcement in the literature. For this, we conducted steps as used in a systematic literature review (B. Kitchenham, 2007) to find literature that could be relevant to architecture enforcement. We use the concepts and categories derived from the interview study in order to label phrases in the publications found during the literature

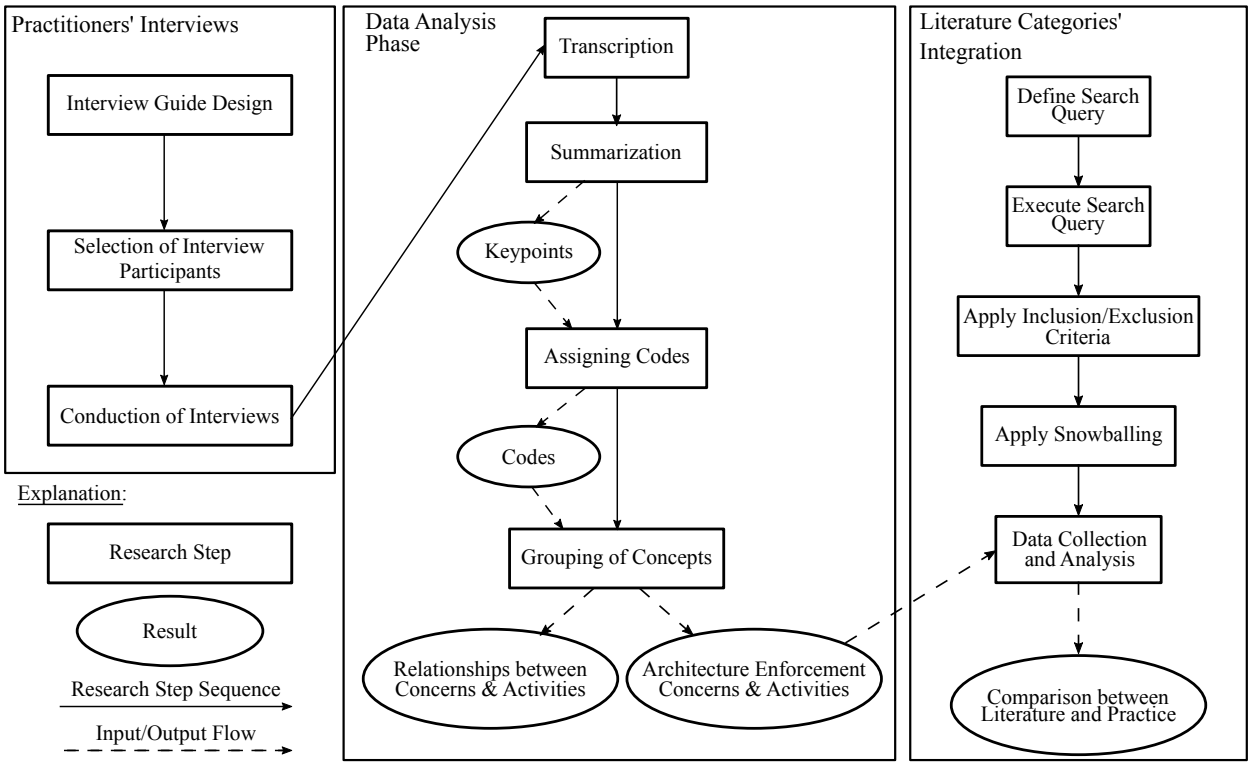

Figure 1: Overview about the overall study design.

collection process. In this way, we examine to which extent the codes and categories derived from the interviews are covered in current literature. The two phases will be explained in the next two sections in more detail (Sections 4 and 5). Figure 1 gives an overview on the overall research process. It depicts the sequence of research steps (boxes) performed in the study and the corresponding results of the research steps (ellipses).

4. Expert Interviews

4.1. Data collection

We decided for a qualitative research method in order to investigate architecture enforcement in practice. For this, we designed an interview study with semi-structured interviews following the guidelines of (Charmaz, 2014). Those interviews are an integral part of qualitative research. They aim to generate new knowledge about a specific topic about which only a few findings exist (Hove and Anda, 2005). It supported us to collect as much new knowledge as possible. An interview guide helped us to keep the focus on our research questions during the interview. Moreover, the guide ensures that the same set of questions were asked in all interviews. The interview guide was designed for a semi-structured interview containing open questions that were chosen according to the research questions. Those questions allow the participants to talk freely about their experience concerning architecture enforcement.

The interview guide contains three parts. The first part classifies the experiences and the technical background of each participant, such as the application domain, years of experience, the development process, the team size, experiences regarding a specific technology etc. In the second part, we asked open questions that aim for identifying architecture enforcement concerns. Correspondingly, the third part contains questions addressing the enforcement activities that software architects perform. We asked the participants which concrete methods they apply in order to enforce architecture.

We tested the interview guide with a master student and a researcher before interviewing the selected participants (see Table 1). The first 12 interviews (A-L in Table 1) were conducted by the first author. The second author conducted

Table 1: List of study participants, their domain and their years of experience.

#	Domain	Role(s)	Exp. (years)	Team size (approx.)	Interview date	Location
A	enterprise (application, integration)	software architect	> 15	100	March 2015	Switzerland
B	enterprise	software architect, consulting	10 - 15	10-20	March 2015	Germany
C	enterprise	software architect	> 20	10	February 2015	Germany
D	logistic	software architect, agile test engineer	10	5-10 per component team	March 2015	Germany
E	accounting / enterprise (migration)	software architect, section manager	10 - 15	50	February 2015	Germany
F	enterprise	software architect, lead developer	10 - 15	2-7	March 2015	Germany
G	enterprise / embedded	software architect, coach	10 - 15	10	March 2015	Germany
H	insurance	software architect, project manager	5 - 10	10 developers, 10 test engineers	February 2015	Germany
I	medical	software architect, software developer	5 - 10	10	January 2016	Germany
J	government / enterprise (application)	software architect, consulting	10	10	February 2015	Germany
K	logistic / enterprise	software architect	5 - 10	5-10 per component team	March 2015	Germany
L	banking, control systems, enterprise	software architect, project manager	> 20	10	February 2015	Germany
M	enterprise	technical leader	10 - 15	10-20	December 2017	USA
N	retail and health-care enterprise	software architect, project manager	> 20	30	January 2018	USA
O	Automotive	project manager	10 - 15	10-20	January 2018	Germany
P	Automotive	software architect	10 - 15	50	January 2018	Germany
Q	Automotive	project manager	10 - 15	10	January 2018	UAE

the other five interviews (M-Q in Table 1). The same interview questions were used. Additionally, the concepts identified from the first 12 interviews were presented to them in order to examine their perspectives and opinions on those concepts.

The detailed interview guide is given in the appendix (see Appendix A). When presenting our study results in the next sections, we will also present some of the questions from the guide and the corresponding responses of the participants.

We targeted experienced software architects from industry as participants of the study. Those architects come from different companies from Germany, Switzerland, and USA. All study participants hold at least a master's degree or a similar qualification in computer science or related fields, e.g. electrical engineering or physics. In total, we interviewed 17 architects from 16 different companies. The interview participants are listed in Table 1. The professional

experience of the participants ranged from 5 to over 20 years, with an average of 13 years. All of them work as a software architect or made significant practical experience in architectural design. For our previous paper (Schröder et al., 2016), we interviewed software architects solely from the enterprise application domain (such as banking, logistic etc.). In this paper, we interviewed five additional software architects. Two of the newly interviewed software architects work in the enterprise application domain and three of them work in the automative domain. By interviewing additional software architects, we, firstly, aim to validate the concepts from the first interview phase and secondly, we especially aim to investigate whether software architects from another domain - the automotive domain - consider other concerns or perform other activities during architecture enforcement.

4.2. Data coding and analysis

In order to analyze the interviews, we transcribed them word-by-word and then followed an inductive qualitative data analysis by conducting open coding (Strauss et al., 1990). Instead of defining codes before analyzing the interviews, we let the categories directly emerge from the data. The only restriction during the coding process was that we explicitly focus on searching for data that may indicate enforcement concerns and activities. However, we did not define a coding scheme beforehand in order to be open minded during the coding process. The results of the coding process are made available on our website[1] for the interested reader. We used *AtlasTi*[2] in order to support the codification process. We performed the following analysis steps:

1) Summarization. In a first step, the main point of the raw data is summarized in a few words in order to raise the abstraction level of the raw data and to generalize the data. This can be only a few words or a whole phrase if necessary.

2) Assigning codes. The main point is then assigned to a code. Hoda and Murugesan (2016) used a similar technique where they first summarize the raw data into so-called key points and then assigned a code to them.

3) Grouping of concepts. Having assigned codes to the data, we continuously compared them with the codes assigned in this and other interview transcriptions. This is called the constant comparison method (Glaser, 1978). Constant comparison helped us to achieve a higher level of abstraction by finding concepts. By iteratively applying the constant comparison method on the emerging concepts, we identify categories. Each category encompasses a group of concepts that appear to relate to the same phenomenon.

The excerpt in Figure 2 shows two examples for the categories ”Architecture Design Principles” and ”Documentation”. The corresponding interview data, key points, and codes are also depicted. As can be seen, the data related to the concept ”Architecture Design Principles” is labeled with the code “loose coupling”. Additionally, we found other phrases in the interviews that were assigned to the concepts separation of concerns, dependencies, modularization and so on. Those concepts were finally grouped into the category “architecture design principles”, that is classified as an enforcement concern. The brackets after a code express properties, dimensions, or other characteristics related with it. This is depicted in the example for the concept ”Documentation”. Instead of simply assigning the code “documentation” to this excerpt, we use the properties in the brackets in order to give a lead that this data is about the amount and the purpose of the documentation in this context.

4.3. Memoing

We also maintained theoretical notes during the analysis of the interviews, so called Memos. They contain notes about the concept discovered during the analysis or first ideas, feelings, and impressions about some concepts. They proved to be very helpful in discovering the enforcement goals and the connections to the respective enforcement activities. We took notes every time we had the impression that there might be an interesting aspect or observation. In order to exemplify this memoing process, we show some notes we have taken for describing observations regarding how the concepts “experiences, skills, programming habits” and “coaching” relate with each other:
The software architect provides guidance for developers regarding implementation of architectural solutions. The

[1] https://swk-www.informatik.uni-hamburg.de/~schroeder/Syso2017/
[2] http://atlasti.com/

Architecture Design Principles

raw data	*...this is also a very important architecture design principle: no coupling. Low coupling and no synchronous communication. Actually, you need to prohibit RMI in Java [...] "*
key point	guideline - loose coupling, no synchronous communication, guidelines - no RMI in Java
code	loose coupling (concern)

Documentation

raw data	*"...we have a very lean documentation, because the running system is more important for us than the documentation. This does not mean that documentation is not important, but we focus on the most essential things...it should be used as a guideline, not a checklist..."*
key point	lightweight documentation, working software over documentation, guiding documentation
code	documentation (amount), documentation (purpose)

Figure 2: Data coding example for the concepts "Architecture Design Principles" and "Documentation"

intensity (property) of the guidance activity depends on the individual architecture skill level in the development team (concept: experiences, skills, programming habits). If the development team needs to implement patterns for example that are new to them, the architect needs to invest additional effort in his guidance activity for example by providing architectural templates and prototypes (enforcement activity), or - if budget and time is available - teaching and coaching the developers in dedicated workshops.

4.4. Axial Coding

We additionally applied axial coding (Charmaz, 2014) in order to find connection between codes. This involves documenting category properties and dimensions from the open coding process by identifying conditions, actions, and interactions with a specific phenomenon and relating categories to subcategories. Axial coding involves the following components:

Causal Condition: conditions that influence the central phenomenon,

Core Category/Phenomenon: the central idea or incident about which a set of actions or interactions are directed,

Strategies: actions or interactions addressing the phenomenon,

Context: in which the phenomenon and corresponding strategies apply,

Intervening conditions: conditions that shape, facilitate or constrain the strategies,

Consequences: outcomes or result of the strategies.

Using this coding scheme, we were able to find relationships between enforcement concerns and activities, so that we can explain which activity is preferably conducted by the architect for a specific concern.
Figure 3 shows an example of axial coding. During enforcement, software architects aim to ensure the agreement on design decisions. This is the core category that is considered during this axial coding example. Several activities can be conducted by the software architect in order to reach this goal. However, the type of activity is influenced by some causal conditions. In this case, those are the programming habits, skills, and experiences in the development team. When enforcing architecture patterns (context), the software architect may adjust the software architecture according to the developers' skill (strategy). This activity is constrained by two intervening conditions: 1) the software architect needs to choose a pattern with which developers are familiar and 2) the pattern still needs to be appropriate for the given functional and non-functional requirements. Applying this activity successfully results in a feasible architecture that can be implemented by the developers and that is accepted by them (consequences).

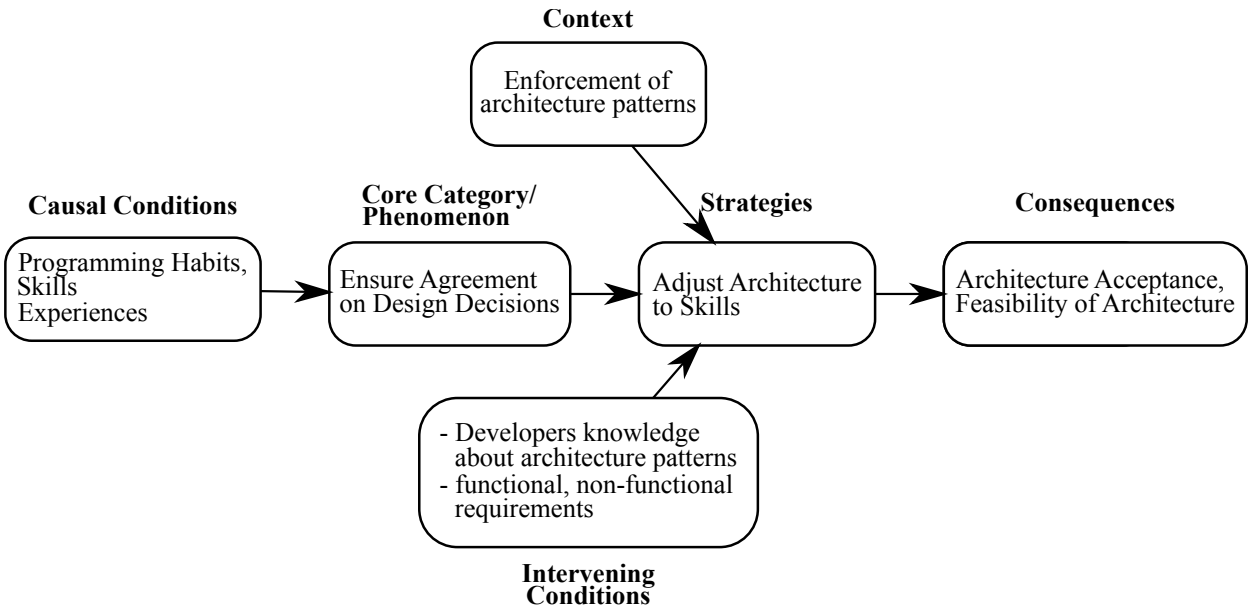

Figure 3: Axial coding example.

5. Literature Review on Architecture Enforcement

We conducted a literature review for finding evidence about architecture enforcement in current literature. We investigated to which extent the codes and categories that emerged from the interviews are covered in the literature. It is worth noting that the literature review is conducted after the interviews. However, we searched for literature independently from the findings of the interviews. We did not use the codes and categories from the interview results in order to formulate the search queries for finding relevant literature. This ensures that we find as many references as possible about architecture enforcement. During the analysis of the literature dataset, we used the codes and categories from the interviews to label phrases in the selected publications. In this way, we investigate which codes and categories are also covered in literature. We did not derive new codes and categories from literature, this means, all findings (codes and categories) emerged from the analysis of the expert interviews as we aim to investigate architecture enforcement in practice.

Our main goal is to focus on empirical studies on enforcement concerns and activities. We adopted practices from the systematic literature review (SLR) process as suggested by (B. Kitchenham, 2007) to make the literature search credible, unbiased, and reproducible. The goal of the review was to find relevant literature that potentially considers and discusses software architect's concerns and activities during architecture enforcement. In the following sections, we describe the necessary steps of the review we have conducted in order to collect relevant literature, namely a) defining the search process, b) defining inclusion and exclusion criteria, c) the data collection procedure and finally d) the data synthesis.

5.1. Search Process

We created a search strategy for the literature search that includes a) a search query term, b) the relevant sources where the publications will be collected from and c) defining the parts of the publications which should be search (title, abstract, full text etc.).

As *architecture enforcement* is not a common term used in software architecture literature, we needed to define this term indirectly in the search query. For this, we used appropriate synonyms to paraphrase *architecture enforcement* or *enforce* (e.g. manage, force, guide, lead) and likewise the terms *concern* (e.g. interest, consider, worry, need, wish) and *activity* (e.g. role, duty, skill, care, task). We additionally added the terms *software architect* and *developer* (and corresponding synonyms) in order to be able to find publications that discuss or consider the relationship between the both, as it is important for architecture enforcement. This then resulted in the following search query:

Table 2: Defined inclusion and exclusion criteria for selecting publications. The criteria are numbered. The prefix I designates an inclusion, the prefix E an exclusion criterion.

Inclusion/Exclusion	Criteria
I1	Publication discusses the role of the software architect.
I2	Publications should define or discuss concerns and activities that could be related to architecture enforcement.
I3	Publications must be written in English.
I4	Publications published in journals, in conferences, book chapters, magazines (also editorials), and workshops proceedings will be included.
E1	The publication such as abstracts, position papers, short papers, tool papers, poster summaries, keynotes, tutorial summaries, conference summaries (or introduction to conference proceedings), workshop summaries or panel summaries are excluded, since they do not provide a reasonable amount of information.
E2	Enforcement considered in the publication is not in the context of software architecture (e.g. policy enforcement, enforcement of law regulations and alike)
E3	Publications full text is not available.
E4	The publication proposes a tool, e.g. for architecture conformance checking.

software architect **AND** (developer **OR** programmer **OR** code **OR** implement) **AND** (role **OR** skill **OR** concern **OR** interest **OR** consider **OR** worry **OR** task **OR** duty **OR** matter **OR** role **OR** need **OR** wish **OR** demand **OR** urgency **OR** essential **OR** important **OR** care **OR** activity **OR** collaborate **OR** communicate **OR** guide **OR** teach **OR** coach **OR** lead **OR** manage **OR** enforce **OR** force)

The Boolean AND operator was used to connect the search terms in order to narrow the search. The Boolean OR operator was used to enable alternative terms and a wider range of search results. It is worth mentioning that not all search engines of digital libraries allow such long search queries (e.g. IEEExplore only allows 15 terms in a query). That is why, we needed to split the query into several sub-queries. The literature is collected for each sub-query. The datasets of the queries were compared in order to find duplicates. The datasets were then finally merged into one dataset. To increase the likelihood of finding relevant data sources, the target of the search query was defined to be applied to the full text and meta data. We applied the search using the query on four well-known digital libraries: SpringerLink, IEEExplore, the ACM Digital Library, and ScienceDirect.

5.2. Inclusion and exclusion criteria

In order to decide whether a publication should be added to the result set, the SLR method requires to explicitly define inclusion and exclusion criteria. They are listed in Table 2. They assess the fitness of the content in each potential relevant publication to the defined research questions. They are applied after all full texts have been retrieved. We decided the inclusion or exclusion of publications based on the title and the abstract. In case we were not sure to include the publication, we added it to the data set for further investigation. In a second phase, the full text of those publications was read carefully in more detail. If relevant information could be found (interests, activities of the architects etc.) the publication was added to the final result data set. We include publications that discuss the role of the software architect and that could be of particular interest in the context of architecture enforcement. Those publications can be empirical studies, but also experience reports from (former) software architects who provide insights in their work and/or give suggestions for other software architects regarding their architecture work. We also decided to include editorials into the dataset, if appropriate. They often provide valuable discussions about the role and interests about the software architect from an experienced architect's point of view (e.g. reports from John Klein, Frank Buschmann, Eoin Woods, Martin Fowler in the Pragmatic Architect Series like in (Buschmann, 2009)), although the reported results are mostly not collected with a rigorous research method. Additionally, we are not interested in papers that only propose a new tool. The focus of those papers is the description of new tools and the evaluation of their effectiveness, e.g. in finding architecture violations or recovering software architectures from code. Those publications do not discuss the role, concerns, and activities of the software architect.

Table 3: Mapping enforcement concerns to publications containing statements that were labeled with the corresponding code.

Concern	Literature
Ensuring Architecture Design Principles	(McBride, 2004), (Lagerstedt, 2014), (Britto et al., 2016), (Buschmann, 2011a), (Caracciolo et al., 2014), (Sherman and Unkelos-Shpigel, 2014), (Sauer, 2010), (Pareto et al., 2012), (Martini et al., 2014), (Staron and Meding, 2017), (Kruchten, 1999), (Woods, 2015)
Appropriate Use of Technology	(McBride, 2004), (Caracciolo et al., 2014), (Buschmann, 2010), (Gokyer et al., 2008)
Aligning with Pattern Characteristics	(Sherman and Hadar, 2015), (Britto et al., 2016), (Martini et al., 2014), (Woods, 2015)
Visibility of Domain Concepts in Code	(Buschmann and Henney, 2010), (Buschmann, 2012)
Visibility of Architecture in Code	(Buschmann and Henney, 2010)
Code Comprehensibility	(Unphon and Dittrich, 2010), (Caracciolo et al., 2014)
Differentiate between Macro and Micro Architecture Decisions	(McBride, 2004), (Christensen et al., 2009), (Clerc et al., 2007)
Adhere to Standards	(Berenbach, 2008), (Groene et al., 2010), (Gokyer et al., 2008)
Ensuring Runtime Quality	(Caracciolo et al., 2014)

5.3. Snowballing

In order to not miss important publications, we performed Snowballing afterwards. We followed the suggestion of (Wohlin, 2014) to apply backward and forward snowballing to increase the likelihood of finding more relevant publications. Snowballing is an iterative method that aims to harvest the references of a paper from the initial result set. Backward Snowballing collects the publications referenced in the reference list of the respective paper. Forward Snowballing identifies relevant papers based on the papers citing the paper that is currently investigated. All papers from the result set of the first stage are investigated in this way. Potential references are eventually added into the final result data set if they fulfill the defined inclusion criteria (see Section 5.2).

5.4. Data collection, synthesis, and analysis

In order to collect the necessary information about architecture enforcement concerns and activities, we applied deductive coding on the publications. We used the categories regarding concerns and activities that resulted from the interviews as the coding scheme. As a supportive tool we used AtlasTi, the same tool that was used for analyzing the interviews. Words, sentences, or paraphrases in the publications that can be related with one or more of the concerns and activities are labeled with a code that is named after them.

As an example, the following statement was mapped to the activity *Adjust Architecture to Developers' Knowledge, Skills, And Experience*: *"The smartest design is basically useless if the development team lacks the skills to implement it properly. Pragmatic architects therefore take explicit care that they design to skill. From all potential alternatives to resolving a specific aspect or concern, they choose the option with which the development team feels most habitable and familiar"* (Buschmann, 2011c). After analyzing all the selected publications in this way, we mapped the concerns and activities to the publications that mention the respective aspect. The results are shown in Table 3 for the enforcement concerns and in Table 4 for the enforcement activities, respectively. Having this information, our interview results are compared with the findings from literature. This gives an impression to what extent the concerns and activities discovered from the interviews are covered in current literature and which evidence is missing there.

6. RQ1: Architecture Enforcement Concerns

In this section, we are going to present the enforcement concerns we discovered from the transcribed interviews. During the analysis of the interview data, we discovered that the concerns can be divided into two categories, namely *Design Decision* and *Implementation Quality* as can be seen in Figure 4. The category *Design Decisions* corresponds

Table 4: Mapping enforcement activities to publications containing statements that were labeled with the corresponding code.

Activity	**Literature**
Modeling Architecture For Developers	(Unphon and Dittrich, 2010), (Antonino et al., 2016), (Tamburri et al., 2016), (Caracciolo et al., 2014), (Martini et al., 2014), (Buschmann, 2012)
Gathering Feedback	(McBride, 2004), (Kruchten, 2008), (Christensen et al., 2009), (Klein, 2005), (Erder and Pureur, 2016), (Galster et al., 2016), (Sherman and Unkelos-Shpigel, 2014), (Martini et al., 2014), (Kruchten, 1999), (Faber, 2010)
Revising the Architecture	(Unphon and Dittrich, 2010), (Faber, 2010)
Correlate Architecture and Code	-
Code Generation and/or Architectural Skeletons	(Berenbach, 2008), (Christensen and Hansen, 2010), (Britto et al., 2016), (Erder and Pureur, 2016), (Caracciolo et al., 2014), (Faber, 2010), (Woods, 2015), (Buschmann, 2012)
Adjust Architecture to Developers' Skills	(Kruchten, 2008), (Klein, 2016), (Tamburri et al., 2016), (Sauer, 2010), (Buschmann, 2011d)
Code Review	(Sherman and Hadar, 2015), (Berenbach, 2008), (Kruchten, 2008), (Unphon and Dittrich, 2010), (Britto et al., 2016), (Christensen et al., 2009), (Tamburri et al., 2016), (Martini et al., 2014), (Kruchten, 1999)
Repository Mining	(Unphon and Dittrich, 2010)
Model-Code-Comparison	-
Automatic Validation of Architecture Rules	(Unphon and Dittrich, 2010), (Lagerstedt, 2014), (Caracciolo et al., 2014), (Martini et al., 2014)
Traceability	(Antonino et al., 2016)
Testing	(Berenbach, 2008), (Paulisch and Zimmerer, 2010), (Erder and Pureur, 2016), (Sauer, 2010), (Martini et al., 2014), (Buschmann, 2011d), (Buschmann, 2011b)

to the most important decisions that architects consider during architecture enforcement. The category *Implementation Quality* encompasses all those concerns that an architect has regarding the correct implementation of those architecture design decisions. Figure 4 shows how the investigated concerns are mapped to their respective category. The concerns will be explained in more detail in the following sections. Each concern is supported with some examples that were mentioned by the participants during the interviews. Additionally, it is shown which concern is mentioned by whom. We structure the results as follows: We first describe the observations made in the interviews and then compare them with findings from literature. Many of the concerns are known from literature, but they were not yet related with architecture enforcement.

6.1. Aligning with Pattern Characteristics

Interviews: Participants mentioned that they aim to ensure that important pattern rules are followed in the implementation. They explicitly differentiate between architecture patterns on a higher level and design patterns on code level. Design patterns are actually not checked, except they are crucial for specific non-functional requirements. That is why, they sometimes validate which design patterns are implemented and if they fit in the specific context: *"which design patterns are used and in which context. Are they only used just because I have seen it in a book or because I wanted to try it or is it really reasonable at this place..."* (code: pattern suitability, Participant C).
The Layer and the Model-View-Controller pattern (Fowler, 2002) where mentioned regularly in the interviews as important patterns that need to be controlled during implementation. Regarding the layer pattern, dependency violations often occur and need to be controlled: *..."when we decided for a layer architecture we took care that the layering is ensured..."* or *..."that I only have the defined relations between the layers"*.

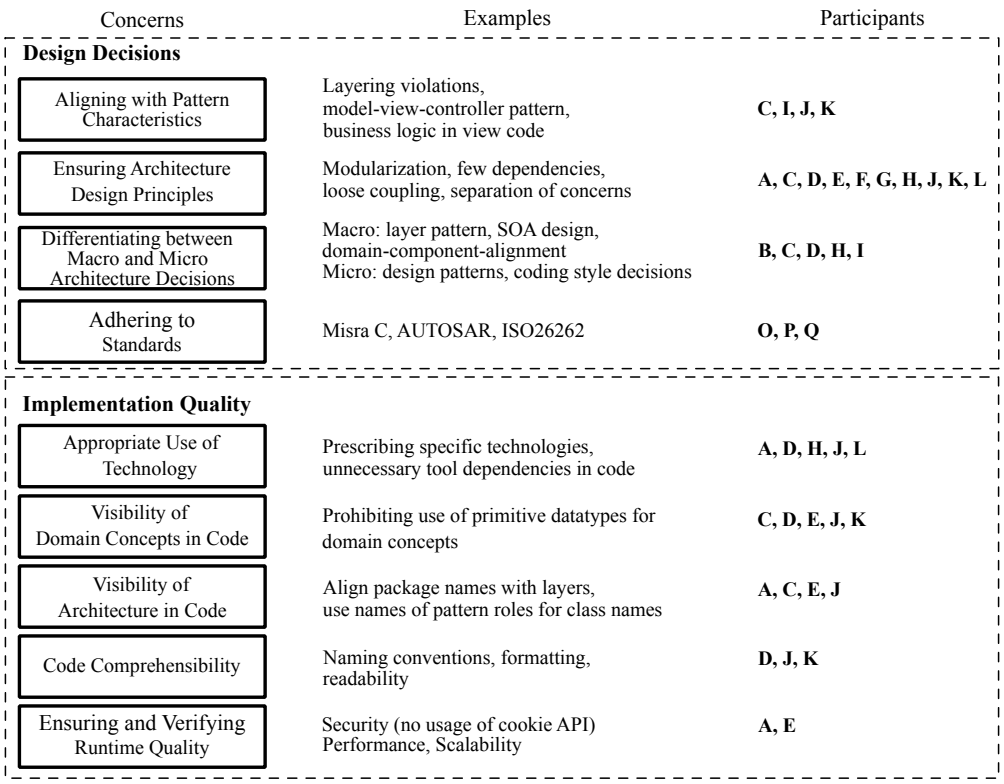

Figure 4: Discovered enforcement concerns, examples mentioned in interviews and corresponding participants who mentioned the concern.

Literature: As stated by (McBride, 2007), the software architect needs to continuously enforce the correct implementation of patterns and needs to able to explain them to other architects and developers. In contrast to our interview results, we found no evidence about on which aspects of a pattern, e.g. allowed relationships between components or pattern rules, the architect should especially focus on.

6.2. Ensuring Architecture Design Principles

Interviews: Software architects especially focus on architecture design principles, such as modularization, separation of concerns, or loose coupling: *"[the system] is composed of very loosely coupled modules that only communicate asynchronously [...] this is also a very important architecture principle: loose coupling..."* (codes: loose coupling, modularization; Participant **L**). Experts also aim to control the dependencies between components, so complex structures and high coupling are minimized or even avoided. Participant **H** especially stated that developers tend to create monolithic structures that are hard to maintain and deploy.

Literature: As can be seen in Table 3, architecture design principles are also considered an important concern in literature. For example, (Caracciolo et al., 2014) investigated in their study, that architects mostly care about minimized dependencies between modules and are considered as *"the main characteristic of a software architecture"*. The same point can be found in (Sauer, 2010) (" avoidance of type interdependencies"). He additionally recommends - based on results from several case studies - that architects should strongly care about principles like information hiding, loose coupling, strong cohesion, design by contract and the open closed principle.

6.3. Differentiating between Macro and Micro Architecture Decisions

Interviews: The interviewed participants differentiate between two abstraction levels on which architecture decisions can be made. We call them macro architecture and micro architecture decisions. Some participants also used other terms like strategic or global (i.e. macro) and tactical or local (i.e. micro). This distinction is used in order

keep the focus on the most important decisions - that are related with the macro level - and to delegate decision with local impact to developers. The participants define the macro architecture as the general idea or ”philosophy”, the ”spirit”, the ”big picture” or metaphor of a system with its most critical architecture decisions). This may encompass the fundamental architectural style, structures, data stores, or communication style: *”...it is important how you regard it. For me there are basically two views about how software is built. First you have the global view [...] There I decide how I design my software, for example using Domain Oriented Design or SOA.”* (code: two different views of architecture, Participant D) or *”...then we have the micro architecture, this is the architecture within each team. A team can decide for its own component for which it is responsible which libraries it wants to use.”* (code: two different views of architecture, macro architecture, micro architecture; Participant K). The interviewed participants report that they basically focus on the macro level of software architecture and consider the micro level as developers' responsibility.

Literature: Christensen et al. (2009) describes two separate views as “architecture in the small” and “architecture in the large” that could be related to both views mentioned by the interview participants. By “architecture in the small” the authors mean aspects that are related to architectural details more close to the implementation. The other view concerns the big picture of the architecture and the quality attributes that drive the architectural design. We found two other publications that use some kind of distinction between different levels of architecture abstraction in the context of architecture enforcement. For example, McBride uses the terms “strategic” and “tactical” in order to differentiate between different level of abstractions (McBride, 2004). He uses this metaphor in order to describe how the architect can manage complexity in the architecting process. By strategic he means “approaches that are high level, broad brush techniques used by architects to master complexity across all audiences.” On a tactical level there are approaches that “address those techniques the architect employs at a lower level of detail, typically with those who construct or use the system.”
However, we did not find any evidence about this distinction in the context of architecture enforcement. In Section 9 we provide a detailed discussion about how the differentiation between macro and micro architecture helps to focus on the most important aspects during enforcement.

6.4. Adhering to Standards

Interviews: In some domains, which involve high risks regarding criticality and safety, the adherence to standards guarantees the achievement of quality requirements and mitigate risks. For example, software development in the automotive domain uses standards for software architecture design (e.g. AUTOSAR[3]). Some standards additionally specify levels, where each level add rules for the design and implementation of software. For example, the ISO 26262 (ISO, 2011) has an Automotive Safety Integrity Level, which specifies the degree of safety. Software architects need to select suitable standards and levels, and make sure that developers adhere to the rules defined by standards during their implementation. One of our interview participants mentioned *”Safety standards are important (...) Not adherence to standards can threaten the life of people”*. Moreover, architects are concerned with adherence to code quality standards (e.g. Misra C[4]). This makes it easier for architects to guarantee good code quality with less effort, because code quality standards could be checked using tools.

Literature: Clerc et al. (2007) report that all products developed at SAP need to adhere to a defined product quality standard. For this, SAP has defined standards that are each classified according to quality topics such as performance, scalability, security, or maintainability. The software architect is responsible for taking all those quality requirements in account and needs to validate them against those standards during the code review.

6.5. Appropriate Use of Technology

Interviews: Some participants mentioned that developers often tend to use a lot of tools and technologies that are not necessary: *”...aim for technologies is the biggest problem. And if you like to use those frameworks because they are providing advanced functionalities, but you cannot control those functionalities if you do not have enough experiences with it...”* (code: aim for technologies, Participant **J**). Due to their complex functionalities and numerous

[3] `https://www.autosar.org/`
[4] `https://www.misra.org.uk/`

ways to be used, it is necessary that software architects also control the way how developers apply a specific technology, since inappropriate use can potentially violate the conceptual integrity and consequently can cause architecture erosion.

Literature: Buschmann (2010) uses the term technology addiction and defines it as *"the use of technology as an end and not a means"*. By this, he means that software architects may choose inappropriate technologies or overuse technologies only because they prefer a specific one. Buschmann states that this is a problem caused by architects. In contrast to Buschmann, architects mentioned in the interviews that this is primarily a problem induced by developers who do not follow the prescribed rules regarding technology.

6.6. Visibility of Domain Concepts in Code

Interviews: Domain concepts relevant for a specific application domain should be explicitly mapped to source code elements as mentioned by the participants. That is why, they aim to use terms for code elements derived from concepts from the application domain: *"...I like to be guided by the domain instead of using technical terms [...] both can work, but from my experience using domain oriented terms is easier to understand..."* (code: domain oriented terminology, Participant **J**). This has several advantages during the development. Firstly, it helps to talk with domain experts about the software design. Secondly, architects and developers can more easily locate the relevant code locations that needs to be change in case that requirements evolve. Another architect emphasized that it *"should be clear which part of the source implements which functionality"* (Participant D), another favored that *"the source code is separated in terms of domains"* (Participant C).

Literature: In Buschmann and Henney (2010) five important consideration are described that should drive the software design in order to make it elegant and effective and consequently more sustainable. There, they describe and define the term "visibility" closely related with this concern we found in the interviews. It is defined as *"the directness and expressiveness of the artifacts that describe and make up software architectures"*. This means that architects are responsible to pay attention to important domain concepts and that they are explicitly mapped in their architecture. For example, in order to implement the concept of an ISBN (International Standard Book Number) an explicit class should be used. Instead of simply using the primitive integer data type or a String, this allows to enforce concept specific rules and contracts and making them explicit.

6.7. Visibility of Architecture in Code

Interviews: Similar to domain concepts, concepts related to software architecture should also be made explicit in the code, e.g. by naming code elements after pattern concepts: *"...therefore it is important that the architecture is recognizable in source code. This is absolutely essential for the structure of the project."* (code: making architecture visible in the code, Participant J). It additionally helps the architect to locate architecture decisions in code in order to validate their corresponding implementation.

Literature: This concern can be related with the consideration "visibility" described in Buschmann and Henney (2010). However, we could not find any study that reports about this concern and how important it is during architecture enforcement. Nevertheless, we are aware that other authors know and care about this problem. For example, Fairbanks describes that there is a gap between architectural models and the source code (Fairbanks and Garlan, 2010). This gap is characterized by the different abstraction levels of both and the different vocabularies they use. The source code acts on the level of packages, classes, functions, and variables, whereas the architecture is described using modules, components, connectors, or ports. As the source code is not able to express those high level concepts of the software architecture, it is cannot be directly expressed in the code. That is why, Fairbanks emphasizes that the architecture should be as visible as possible using the means provided by the programming language.

6.8. Code Comprehensibility

Interviews: Inconsistent use of naming conventions and coding styles decreases code comprehensibility. Participants mentioned that incomprehensible code significantly contributes to architecture erosion. If the code cannot be understood by other developers, this may lead more likely to architectural violations: *"if you strictly follow this*

approach then you have very readable code. From my experience, readable code tends to be more stable. This means, it is easier to implement code that is conform to the architecture and does not have any [architecture] violations..." (code: code comprehensibility, code comprehensibility supports architecture conformance; Participant J).

Literature: Evidence about this concern related architecture enforcement was only found in (Caracciolo et al., 2014) and (Unphon and Dittrich, 2010). In the latter, it is reported that architects are aware to enforce a good naming of classes, methods, or interfaces so that it helps developers to understand the corresponding concept behind a specific code element. Caracciolo et al. (2014) found that architects define and enforce specific naming conventions to guarantee code quality.

6.9. Ensuring and Verifying Runtime Quality

Interviews: Interviewed participants mentioned that they are concerned with architecture significant requirements (ASRs) related with security, performance, and scalability. That is why, they aim to ensure that there are no ASR-violating code statements in the implementation: *"then you investigate the code and validate if it fulfills the ASRs. [...] if elements from the domain model, such as orders or credit card information, are stored in a cookie, then this is a violation obviously regarding the decision "Data-based or Server Session State". This has highest priority and needs to be repaired immediately."* (codes: ASR violating code structures, practice - checking ASRs; Participant A).

Literature: The study of Caracciolo et al. (2014) also shows that software architects actually are concerned about quality attributes and about enforcing them. They investigated which quality attributes are considered most important and how they are enforced by the architects in practice. This results in a concrete list of prioritized quality attributes and corresponding measures.

7. RQ2: Architecture Enforcement Activities

In the following, we present the results of the second research question by asking the open-ended question *"How do you ensure that your architecture and your concerns are implemented as intended? Do you follow any strategies?"* and others. The result is a collection enforcement activities as depicted in Figure 5. We categorized the activities according to the enforcement goals *Ensure the agreement on design decisions with stakeholders* (abbreviated as *Ensure agreement*) and *Check the conformance between implementation and design decisions* (abbreviated as *Conformance Checking*). In the following, we present each enforcement activity category and the corresponding activities and support them with statements from the interviews.

7.1. Achieving Mutual Understanding of the Architecture

We define this category as the activity that aims to achieve the coherence about the concepts of the software architecture in the development team. The software architecture is kept in the mind of the developers and the architects. All of them should have the same understanding and picture about the architecture and its underlying decisions: *"a common picture - keyword modeling - is very important here, to have a starting point and to have it started in the same direction"* (code: common understanding of architecture, using models for comprehension, Participant **B**). If a shared understanding is missing in the team, it is more likely that the architecture is (unintentionally) violated by developers.

Developers should also have a common understanding about the prescribed architecture, its rationale and its goals that have to be achieved with it: *"skilled people do automatically know how they implement code that is conform to the architecture, because they know, why it should be like that. Then - without help - developers have the architecture in their mind and recognize if architectural goals are fulfilled or not."* (codes: architecture awareness, personal quality; Participant B). The risk of introducing architecture violations increases when developers are not aware of architecture goals.

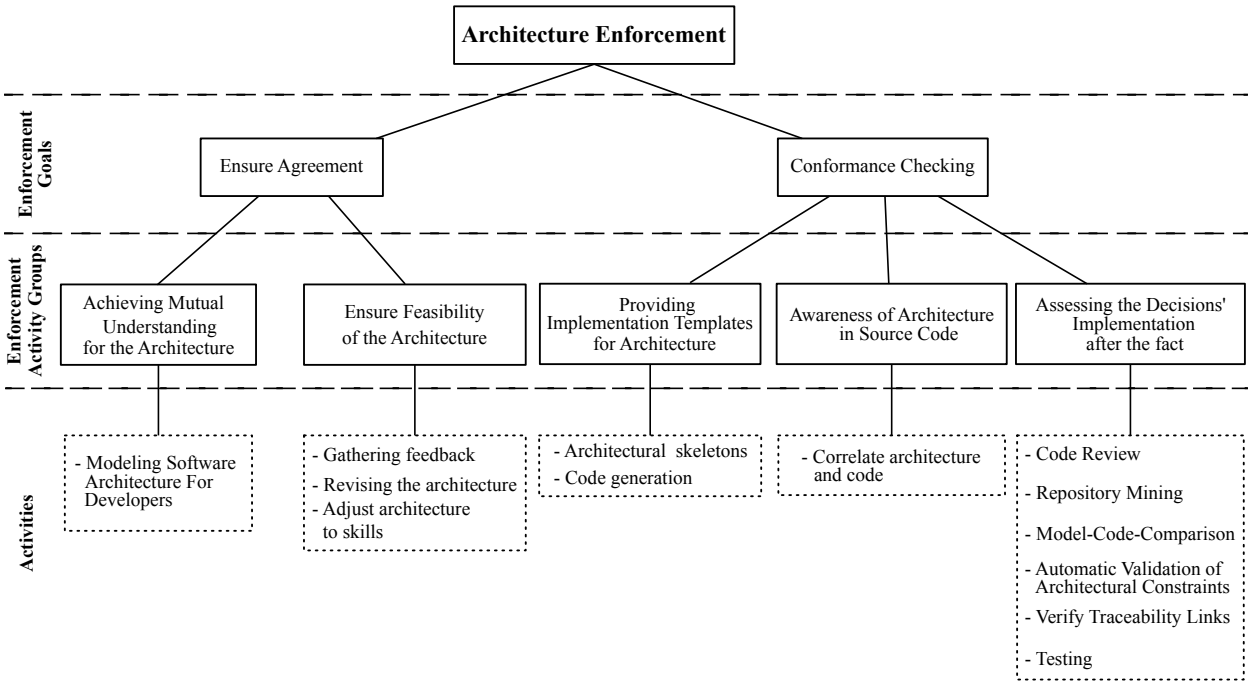

Figure 5: The mapping of the identified categories of enforcement activities to the respective enforcement goal.

7.1.1. Modeling Software Architecture For Developers

Interviews: Architecture models greatly help to achieve a common understanding about the architecture. This activity entails two steps: a) selecting an appropriate modeling notation (formal to informal) for describing the software architecture and b) the actual modeling of the software architecture. Visualizing software architecture with models is a common activity performed by software architects in order to achieve a mutual understanding and should be considered as an integral part of enforcement. The modeling notation may be formal or informal. For example, participant **B** utilizes diagrams from the Unified Modeling Language (UML) (Rumbaugh et al., 2004) in order to explain how systems should exchange messages with each other: *"...the most important diagrams are activity diagrams and sequence diagrams. If developers asked us how it should be implemented, we used the diagrams in order to explain them the specification and make them understand the planned architecture..."*. Participant **L** uses a more formal modeling language - Fundamental Modeling Concept (Knöpfel et al., 2005) - in order to support the architecture understanding process. He emphasizes that models are important for adequately communicating and discussing with each other about an architectural design. He argues, that it is crucial to have a picture about the planned software system that everybody in the team can understand. This can only be supported by using proper visualizations: *"...the developer needs something that is written down in terms of visual models he can work with. Something that explicitly shows the most important components and processes..."* (code: models and visualizations). Moreover, he reported that human communication without supportive visualizations embeds ambiguous interpretations. Therefore he prefers to use models to resolve those ambiguities.

Some participants state that informal whiteboard drawings are the most effective and appropriate way to visualize the current architecture, since those drawings are commonly kept in the developers' office and are therefore always present: *"...it is more important that the picture [about the architecture] is kept on a whiteboard in the development team's office, so that the architecture is kept in their sight..."* (codes: whiteboard drawings, shared understanding, documentation(type), Participant E).

Literature: Unphon and Dittrich (2010) reports that architects often use informal drawings like boxes-and-arrows or UML-like diagrams for visualizing and explaining the overall architecture and in order to ensure that developers have an understanding about the architecture. Caracciolo et al. (2014) also observed that architects sometimes use

more detailed activity diagrams for modeling an message exchange protocol. In the other publications we have found the use of models and visualizations is also promoted, however they do not describe how models are used and which types of architecture models are important for architecture enforcement.

7.2. Ensure Feasibility of the Architecture

The software architect needs to make sure that the developers are able to realize the software architecture by ensuring its feasibility. By doing this, he reduces the risk that the developers might not accept the software architecture as it was designed by the software architect. The architect should always be *"... anxious for getting the architecture accepted by the developers and that they [the developers] want to implement it this way."* (codes: encourage acceptance of developers for architecture, willingness; Participant **B**). In the following sections, we present three activities that help to ensure the feasibility of architecture.

7.2.1. Gathering Feedback

Interviews: The interviewed participants aim for regular discussion with the development team, especially in order to address crucial architectural violations or to get feedback from the team regarding the current architecture design. To make this possible, the architect has to be available for feedback as developers might not agree with the architecture solution or the design does not fit the current requirements anymore: *"...we had regular meetings with the developers and showed the developers where deficiencies in the architecture are and where rules were violated."* (code: discussion of violation, Participant **E**).

Literature: As can be seen in Table 4, gathering feedback is also considered an important activity. For example, Klein stated that it is important for the architect "to develop a relationship with the development team that will encourage the developers to include the architect in discussions and decisions." (Klein, 2005). Faber additionally emphasized that it is crucial to not dominate the developers in this process and eventually accept architecture deviations if developers can justify them (Faber, 2010); a finding that we can also confirm.

7.2.2. Revising the Architecture

Interviews: Participants reported that a software architect needs to be open for potential revisions on the current architecture design. If an architectural solution is too complex to be implemented, the software architect needs to find another solution suitable for the given requirements and simple enough so that it can be implemented by the developers: *"...for example they [the programmers] said that it was not possible to do it differently, because this and that was too complicated, so that we adapted the architecture rules in consequence and said it has to be different here actually, but otherwise it is too complicated."* (code: solution too complicated, revise architecture, Participant **E**).

Literature: We found two publications that implicitly mention this activity (see Table 4). Unphon and Dittrich (2010) use the term "walking architecture" to describe a role mainly responsible for maintaining and "realizing changes in the architecture" if necessary.

7.2.3. Adjust Architecture to Developers' Skills

Interviews: The software architect should choose architectural solutions (e.g. patterns, technologies) based on the individual skills and experiences in the development team in order to get the solutions accepted. In this way, he can ensure that developers are more familiar with the concepts and to reduce the risk of architecture violations.

Literature: Klein considered this activity especially important in the initial designer role of the software architect (Klein, 2016). According to Klein, the architect must consider how the architecture should be implemented. This strictly requires to assess the team's skills and experience and consequently adapt the design correspondingly to those skills and experiences. This supports the developers to appropriately implement the architecture and stick to the conceptual integrity of the system.

7.3. Providing Implementation Templates for Software Architecture

Architecture design decisions should be implemented in a uniform way by every developer. The developer's skill and experiences, e.g. from previous projects, and his programming habits influence to a great extent how he implements architectural design decisions. Due to his habits, he can potentially violate those decisions: *"...leaving it to the developers is not suitable since every developer has a different background and experiences. When I just tell them that they should start with programming, then this leads to chaos..."* (code: programming habits and experience of developers; Participant L). That is why, the software architect is responsible to provide guidance in the implementation of the decisions. We found several activities that were conducted by software architects to address this aspect.

7.3.1. Architectural Skeletons and Code Generation

Interviews: In case that architectural concepts are new to developers and the architecture cannot be appropriately adjusted to the team structure, the architect is responsible for coaching and supporting developers adequately. The coaching phase is conducted until all team members are able to implement an architectural solutions. The architect can provide skeletons (or architectural templates) in order to support the coaching process. Using code skeletons they guide how a specific decision has to be implemented or a specific technology has to be used. Architectural skeletons provide a reference for the developers during the implementation: *"...you build something as an example and present it to the developers..."* (code: architectural templates, Participant **A**). Those templates need to be built precisely and carefully according to architecture design decisions and state-of-the-art best practices, as emphasized by participant **A**. Otherwise developers could unintentionally violate underlying decisions. Another way to provide guidance is to generate those templates via code generation. The skeleton code contains the main structure for a single component and the glue code for connecting the component with another one. Developers are forced to stay within the boundaries of the generated code and are not allowed to break the generated code, otherwise important decisions cannot be guaranteed anymore: *"...and then using xslt transformation a skeleton is generated from the xml that is given to the developer and we generate skeletons for the test team. The developer is enforced to work with this skeleton."* (code: code generation, Participant **H**).

Literature: The use of architectural skeletons, templates, or prototypes seems to be a common and accepted method to explain and exemplify the software architecture, as suggested in (Berenbach, 2008), (Christensen and Hansen, 2010), or (Faber, 2010). Caracciolo et al. (2014) also found that some architects use code generation to simplify the creation of modules.

7.4. Awareness of Architecture in Code

Developers should be aware of when they change code that implements important architecture design decisions. It is the software architect's responsibility to show the developers what are the most crucial code parts regarding architecture design decisions: *"...we showed the developers where [in the code] the architecture rules are actually violated..."* (Participant E).

7.4.1. Correlate Architecture and Code

Interviews: Software developers should be aware whether they are changing architecture-related code. Architecture-related code may be a code part that is responsible for an architecture pattern or tactic implementation. Those parts are significant as they address important quality attributes. In case that architecture violations occur in this specific code part, those quality attributes cannot be guaranteed anymore. That is why, participants emphasized that it is important to clearly describe how architecture and the related concepts are implemented. By doing this, the implementation of architectural solutions can be located easier. For example, participant **A** stated that the layer pattern should be clearly mapped onto the package structure. Additionally, classes participating in an architecture pattern implementation should clearly be named after the pattern role. This activity is strongly related with the concern "Visibility of Architecture in Code" described in Section 6. This was also favored by Participant J who stated that *"...developers are able to orientate themselves easier in the code..."*.

Literature: We are aware that some approaches exist allowing to increase the visibility of software architecture in the code. However, those were not found during the literature search. This may be a result of our defined inclusion

and exclusion criteria. Nevertheless, some of those approaches should be mentioned here: The architecturally-evident coding style is a predominant method to make architecture explicit in the code (Fairbanks and Garlan, 2010). Simon Brown developed a framework - called Structurizr[5] - helping to make architecture concepts like components or containers explicit in source code. ArchJava (Aldrich et al., 2002) aims for bridging the gap between architecture and implementation embedding a formal architecture specification into Java source code using additional language elements that represent components and connectors. Those elements allow to enforce communication integrity (Aldrich et al., 2002). However, we did not find any evidence in the dataset collected during literature search whether this technique is applied by software architects in order to enforce the software architecture.

7.5. Assessing the Decisions' Implementation after the fact

During the interviews we asked all participants the following question: *"What are the specific steps you perform when you inspect the source code in order to assess the implementation of the architecture decisions?"*. This results in a list of categories describing common methods and activities for assessing the decisions' implementation.

7.5.1. Code Review

Interviews: In order to assess the implementation according to architecture design decisions, the interviewed participants mostly rely on manual code reviews. One architect stated that this activity *"is similar to the comprehension process of a developer who is new in the team and tries to understand how the software system works. But developers and architects have different goals during this process each. The developer mainly wants to implement new features, while the architect wants to check architecture conformance"* (participant **C**). During this activity they use their mental model about the software system as the guideline: *"...a picture about if the components are appropriate, if the modules are implemented according to how it was intended..."* (Code: expectation about intended design, Participant C). In this process, software architects often ask questions about the observed software systems that entail exploration and navigation, such as who implemented this component and where a specific feature, architectural pattern, design pattern, technology is implemented or used. It is then evaluated informally if an implementation roughly represents this mental model. During this process, code analysis tools can be used as a source of information: *"...what you can do is, you run a code analysis tool and then you are looking at the spots that are interesting..."* (code: finding hot spots, results from code analysis tools as first impression, Participant **K**). Using analysis tools showed to be quit used to verify the adherence to standards (e.g. MISRA C). It should be noted that not all interview participants conduct code reviews on a regular basis or are not involved in code reviews due to other duties in the project. Moreover, code reviews are costly activities. Nevertheless, most of the participants emphasized the importance to be informed about the current state of the source code which is achieved by code reviews.

Literature: Code review is a valuable method in order to manually evaluate software quality (Baker, 1997). Basically, there are two categories of code reviews: inspections (Fagan, 1999) and modern code reviews (Bacchelli and Bird, 2013; Kollanus and Koskinen, 2009). In literature, it is commonly accepted that the architect should also participate in code reviews in order to assess the implementation of his architecture design as suggested in (Britto et al., 2016), (Christensen et al., 2009), or (Martini et al., 2014). Further references can be found in Table 4.

7.5.2. Repository Mining

Interviews: Repositories and review systems, such as Gerrit[6], provide useful information about which changes are made on the software system. One participant mentioned that he uses a review system in order to assess the implementation step by step by reviewing single commits or pull requests of developers. He hereby especially focuses on architectural issues. Using history information, he can easily investigate *what type of changes* were conducted on a set of classes and especially *who* did the change. Moreover, introduced architectural violations can be traced back to their emergence. The steps of the implementation can be reproduced and rationale about specific code-level decisions can be reconstructed.

[5] https://structurizr.com/

[6] https://www.gerritcodereview.com/

If the architect knows about the individual skills in a team, he can focus source code inspections on changes by developers that have less skills, are inexperienced, or are new to a project: *"...you know basically who works on which parts, this means if I know from experience that I have to have a closer look on what he or she has created then it is possible that I have to inspect each class [...] because he or she can create an unusual solution on the most unobtrusive parts"* (code: focused inspection based on individual skills of developer, Participant **C**).

Literature: Repository mining is a well-known technique in software evolution research (Thomas et al., 2014). For example, it is used to detect hidden dependencies between modules by evaluating so-called change coupling (Gall et al., 1998). There are several approaches that investigate developer collaboration networks in open source systems, e.g., to find experts and recommend them for code reviews automatically (Thongtanunam et al., 2015). (Unphon and Dittrich, 2010) reports that, where the software architect reviews single commits and the corresponding changes made by the developers. Those changes are then discussed during code reviews. However, we did not find any evidence about how software architects exploit the individual skills of the developers in order to focus the assessment of the decisions' implementation.

7.5.3. Model-Code-Comparison

Interviews: We asked the participants to which extent architecture documentation and models are used in assessments. Some experts (**B**, **I**, **J**, **L**) use documented diagrams and models for conformance validation between the implemented software system and the architecture. UML class diagrams, sequence diagrams or component diagrams are commonly used. The participants compare those diagrams with models that are automatically extracted from the underlying implementation. The comparison is, however, performed manually. A possible evaluation scenario includes the validation of the message exchange between components and if it conforms to the prescribed behavior given by the documented UML sequence diagram.

Literature: There are a lot of approaches allowing the recovery and reconstruction of software architecture (Ducasse and Pollet, 2009). For example, they are able to discover class diagrams, find components/modules or even reconstruct sequence diagrams using static and/or dynamic analysis (Lungu et al., 2014). These approaches can be used to manually compare the implemented architecture with the intended architecture. However, regarding its application in practice and in the context of enforcement in particular, we did not find any publications reporting about how software architects use this technique or which support they require in this activity to efficiently perform architecture enforcement.

7.5.4. Automatic Validation of Architectural Rules

Interviews: We also asked architects to which degree they formalize architectural aspects in order to allow a formal validation of a software architecture. We found that participants rarely formalize architectural rules. Some of them formalize the layer pattern and validate if implemented layer dependencies adhere to the prescribed ones. For this, they use tools such as Sonargraph: *"...actually, automated validation of the macro architecture is conducted with the Sonargraph tool. This is about checking the defined relationships between layers and slices."* (codes: ensuring macro architecture, automatically validate layer pattern; Participant J). Some participants also mentioned that they formalize rules on a lower level of abstractions. For example, thresholds for complexity metrics amongst others are validated automatically with code quality tools like Sonarqube or Checkstyle. It is worth mentioning that not all architects relate those low-level rules to architecture, but rather to a good programming style. However, some architects aim to be responsible for both: *"...in the strict sense, this is not really architecture, but I think it is better to manage it together [architecture and coding rules] ... the naming convention was given by us [the architecture team]. This is not really architecture, rather programming guidelines, but I think they belong together."* (codes: restricting complexity metrics, naming conventions; Participant E).

Literature: Lagerstedt (2014) reports on his experiences as a software architect. He used automated tests to verify rules and guidelines related to non-functional requirements, e.g. in order to ensure that important dependency rules are not violated. Caracciolo et al. (2014) also reported that not only dependency constraints are validated automatically, but also more low-level constraints that concern the coding style or naming conventions.

7.5.5. Verifying Traceability Links

Interviews: To verify the implementation of design decisions, software architects ask developers to ensure up-to-date traceability links between software architecture design decisions and their implementation. To achieve this, architects and developers use traceability tools (e.g. Reqtify[7]) to link architecture significant requirements and their corresponding implementation modules. The tools have the ability to parse several documents, and link sentences from different documents to create a traceability tree. In projects that involve different industrial partners, traceability becomes very important, because possible failure and their consequences could be traced and legally proofed. For example, manufacturing of cars involve many different industry partners, each being responsible for one or more components. In case of a problem at a certain component, problems are traced and the responsible partner is identified.

Literature: Antonino et al. (2016) report about challenges and recurring problems of software architecture design in the embedded system domain. There, embedded architects normally lack knowledge about software architecture design, related practices and principles. They emphasize that maintaining traceability between different artifacts is of paramount importance in this domain. However, they observed that *"embedded- software architects neglect the fundamental traceability that should exist between the architecture drivers and architecture design"* (Antonino et al., 2016).

We did not find further evidence in the publications about traceability in the context of enforcement and how software architects actually use traceability in the enforcement process. However, we are aware of some traceability approaches that establish traceability links between architecture and code. Those approaches intent to help developers to understand the architecture design and its corresponding implementation. Approaches that establish traceability aim to support identifying the correct links between architecture and implementation artefacts and to ensure consistency between them. For example, in order to control erosion of architecture decisions' implementation, the approach proposed by Mirakhorli and Cleland-Huang (2016) allows tracing architectural tactics (Bass et al., 2012) to architecture-relevant pieces of code and warns the developer if he changes code of this significant part. Buchgeher and Weinreich (2011) proposes an event-driven traceability approach of that establishes traceability links while developers implement a certain design decision.

7.5.6. Testing

Interviews: Software architects use tests in order to ensure functional and non-functional requirements. They aim to ensure enforcement concerns using appropriate tests . Participants aim for a high test coverage in order to help to discover architectural violations: *"...in case there are only a few tests, then it is likely that people do not build it correctly. This leads to incomprehensible code and consequently to architectural violations."* (test coverage supports architecture conformance, Participant **J**).

Literature: Sherman and Hadar (2015) investigated the role of software architects in practice. In their study, they observed that software architects also use tests to document and illustrate the architecture design. Lagerstedt (2014) reports his experiences about his work as an architect. He uses automated tests in order to verify non-functional requirements and emphasizes the importance of automated tests with fast feedback. He also applies this technique as an integral part of his architecture work. Buschmann (2011d) proposes that the architect is responsible for ensuring that component contracts should have a corresponding unit test in order to be able to validate the component's behavior.

8. RQ3: Connecting Enforcement Concerns and Activities

In the last two sections we considered enforcement concerns and activities separately. However, we observed connections between those two in the interview data. This means that software architects prefer specific activities for corresponding concerns. For example, static analysis tools are preferably used in code reviews (the activtiy) for evaluating code comprehensibility (the concern) by assessing the adherence to naming conventions. Figure 6 illustrates those relationships. They are labeled with roman numerals. In the following, we describe the relationships between concerns and activities and support them with corresponding data from the interviews. For each description

[7] `http://www.claytex.com/products/reqtify/`

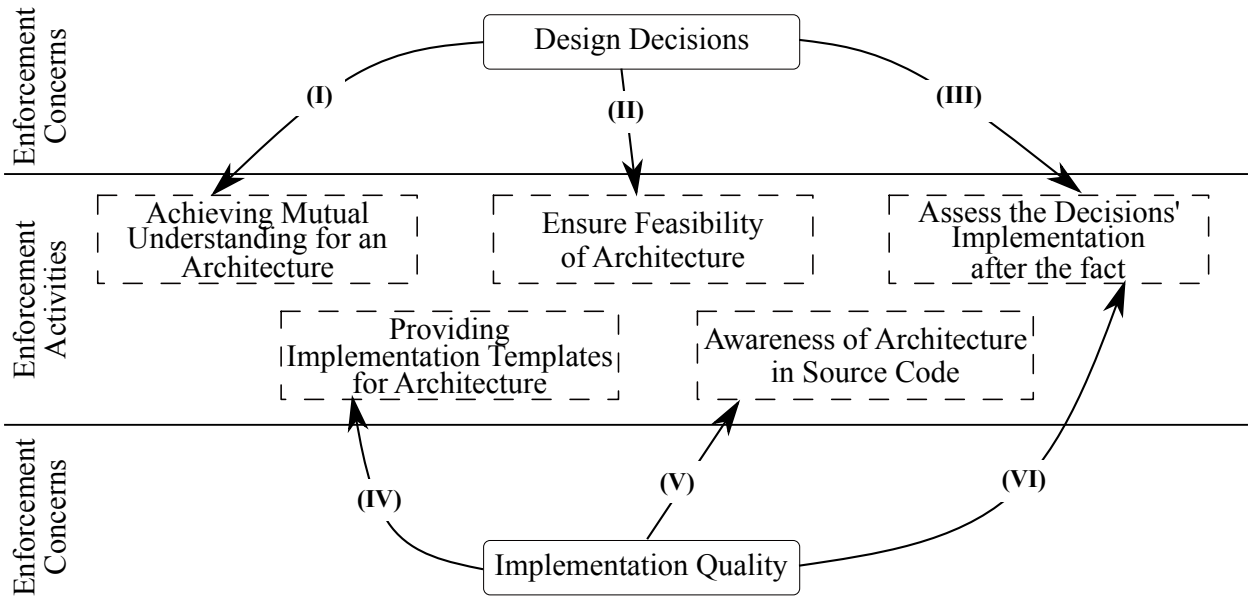

Figure 6: Relationships between Enforcement Concerns and Activities.

we refer to the link by using the corresponding roman numeral. We again distinguish the two categories of enforcement concerns *Design Decisions* and *Implementation Quality*.

8.1. Design Decisions

(I) Understanding Design Decisions. Architects apply architecture modeling in order to visualize and explain the main components or patterns to make the developers understand the architecture and increase the shared understanding about the architecture in the development team: *"...models are very important, because humans need something visual, something that can be looked at and that can be understood. Everybody has the same picture in their head, because the same picture lies in front of them on the table. [...] We created a big picture about how the server is composed of components and layers..."* (participant **B**).

(II) Ensuring the feasibility of Design Decisions' Implementation. The software architect needs to ensure that developers are able and are willing to implement an architecture decision by gathering feedback on the decisions, revising them, or by adjusting the decisions according to the developers' skills. For example, the software architect may choose specific patterns that are well-known by the developers in order to increase the architecture acceptance: *"...talking with the developers and examining which reference architectures do they know, do they know the layer architecture or more complex patterns [...] then you take this pattern and try to apply it. This works very well, because the developers feel more comfortable [...] and if you find any violations then they will rather accept them and repair them. Because, they accept this architecture and they want to preserve it. This is much more crucial than having a complex architectural design..."* (participant **J**).

(III) Validating Design Decisions. As described in Section 7, software architects use several methods in order to validate the decisions' implementation. However, they choose a specific method that depends on the type of decision they aim to validate. Architects may use architecture analysis tools like Sotograph in order to evaluate if the layer pattern is implemented correctly: *"...I validated the layer model with the architecture visualization tool - the Sotograph - with this you can perfectly create the layer architecture, in the code we explicitly labeled the layers using a specific naming convention so that it can be directly mapped to the Sotograph. The tool showed us the different violations..."* (participant **E**). However, regarding other macro architecture decisions that cannot be easily checked with tools, like for example the architecture design principle "separation of concerns", architects conduct manual code reviews: *"...what can not be validated with tools or cannot be validated automatically is*

if a specific functionality belongs to this layer or not or to another layer. This can be only examined through code reviews. And this is something that we have validated through code reviews." (participant **E**).

8.2. Implementation Quality

(IV) Ensuring quality of decisions' implementation using templates. The concern *appropriate use of technology* (see Section 6) is addressed by enforcing a specific way of how a technology or library is to be used. This needs to be enforced in order to address the programming habits of developers who may have different ways of using a specific technology: *"...JMS can be used in six, seven, eight different ways. Can this be decided by the developers? No, it cannot, because everybody uses it differently. That is why we built a framework [...] and we enforced that they only used this framework."* (participant **L**).

(V) Acknowledging developers about architecture relevant source code. In case of visibility of architecture in code another software architect enforced that the layer pattern was explicitly mapped to the package structure of the code base and corresponding terms from architecture patterns are used for classes that reify architecture pattern roles. With this enforcement activity, he aims to increase the architecture awareness of the developers during coding: *"...the layer architecture must be clearly recognizable in the package structure and you must clearly see where each subsystem is located. [...] projects that are structured in this way helps the developers to orientate themselves in the code easily and to recognize the architecture..."* (participant **J**).

(VI) Ensuring Code Quality. By performing conformance checks, it can be validated if the implementation quality is ensured in the code according to the concerns described in Section 6. Software architects may use static code analysis tools in order to evaluate the overall code quality regarding code comprehensibility (*concern*) by investigating code metrics (*activity*): *"...we use tools that are able to find problems on a code level, like for example big classes with too many methods, method call depths, bad class names and so on [...] this is what is necessary in order to be able to ensure that the code remains clean during changes..."* (participant **B**). Implementation quality can be enforced by defining specific naming and code structuring conventions, so that the visibility of domain concepts can be ensured. For example, an interviewed participant reported that he strictly prescribes that no primitive datatypes are allowed for domain concepts. In this way, he addressed the programming habits of developers who tend to use only primitive datatypes like int or String to model domain concepts: *"...I started in this project and nobody has ever heard this concept. Never. Everything is modeled as an int, maybe bytes, [...] and you can see that this is not a real type-safe system. [...] they do not build their own data types [...] That is why I enforce that there are no primitive data types at public interfaces [...] they are not thinking about it and simply transform data from one place to another although they are not allowed to. This is indeed a problem."* (participant **J**).

9. Discussion and Research Hypotheses

This section discusses the findings and the implications for practitioners and academia. The results from the expert study combined with the results from our literature review indicate that there are still gaps between enforcement methods from practice and from literature.

9.1. Comparing Practice and Literature

A new contribution of our paper is the comparison between enforcement approaches from practice and literature. With this comparison we aim for a comprehensive overview about architecture enforcement comprising. This comparison has shown that further research is necessary in order to fully understand the enforcement process and to provide supporting methods and tools. We exemplary choose and describe some of the concepts that may be interesting to be investigated in more detail.

9.1.1. Enforcement Concerns

Macro and Micro Architecture Decisions. We think that the distinction between macro and micro architecture should be defined more clearly by the software architecture community. Currently, the boundary between macro and micro architecture is still defined very vague. A more clear distinction could help architects to focus on the most important decisions during architecture enforcement. We claim that we made a contribution to this distinction which could support architects by setting a scope for architecture decisions and to determine decisions that are the responsibility of the developers.

Visibility of Domain Concepts in Code. Software architects are concerned about how domain concepts are mapped to the code. Domain models are typically not considered to be related with software architecture as stated in (Fairbanks and Garlan, 2010), but as confirmed by the interview results, this aspect should not be treated less important. We propose that new approaches for architecture enforcement should also consider domain models.

9.1.2. Enforcement Activities

Model-Code-Comparison. Although knowing that a lot of approaches exist in order to reconstruct software architectures, class diagrams, or sequence diagrams from code, we did not found any studies in our literature search that report about how those approaches are actually used by software architects. That is why, we think that further studies are needed in this direction. This is necessary, in order to provide methods and tools that effectively support the software architect in his enforcement activities.

Verifying Traceability Links. We could find only one publication in our literature result set reporting about experiences of software architects using traceability approaches. Traceability approaches could additionally help to connect the software architecture with architecture relevant code. This additionally supports the activity "correlate architecture and code".

Formalization of Architectural Rules. We found that formalization of architecture patterns is rarely used by architects although a lot of approaches are proposed in literature. Static dependencies are often used as a main criteria to define architectural rules. Rules concerning layer dependencies are validated regularly, whereas other types of architectural solutions, e.g. Model-View-Controller-Pattern, architectural tactics or communication styles, are not validated. Further studies in this direction are necessary. In a future study, one might explicitly ask about which pattern constraints are considered during enforcement and how they are actually enforced (which tools architects use etc.). Another direction for future work would be to investigate which problems they face with current conformance checking tools and which support they need for conducting the enforcement.

9.2. Research Hypotheses

Hypothesis 1: Enforcement concerns and activities depend on the domain for which the software architect designs the software architecture.

During our interviews with participants, we found differences between domains regarding their enforcement concerns and activities. Some enforcement concerns and activities appear to be important for architects at certain domains. In the automotive domain, we found that adherence to standards and performing traceability are paramount for software architects, because they support achieving quality attributes and mitigate risks regarding safety. Both standards and traceability have not been mentioned by other interviewees, who work in developing enterprise software. On the other hand, some enforcement concerns show to have lower priority for architects in the automotive domain. One participant mentioned *"using design patterns is not common in automotive"*. Thus, architectural patterns show to have lower priority in the automotive domain than in enterprise applications. Another participant mentioned *"In embedded systems, we do not adjust software architecture to fit developers' skills, because hardware design constraint our software design"*. This shows difference to enterprise software, where architects could adjust software architecture to fit skills of developers as part of their enforcement activities. In automotive software, it is not usual to try several technologies as in enterprise software. This is because releasing software for automotive has several legal restrictions.

One participant mentioned *"Legal consequences make messing inside the source code limited"*. The differences between both domains (automotive and enterprise software) come from the different priorities for quality attributes, and constraints. In embedded systems, memory consumption and performance have higher concern than other quality attributes, while in automotive, safety is paramount.

Hypothesis 2: Enforcement concerns and activities depend on organizational factors like the software development process, budget constraints, or other duties the software architect has in the project.

It is worth noting that we discovered different factors that may impact the enforcement activities conducted by the software architect. For example, the extent to which enforcement activities can be conducted may be constrained by other duties a software architect has in a project. For example, some architects are responsible for various tasks that also encompass project management. Those architects rather classify themselves as "non-technical", high-level architects (or *architecture astronauts* as defined by Buschmann (2009)) who do not participate in coding activities or even do not conduct a code review. They delegate this task to so-called lead developers who participate in coding, but additionally own a high-level overview about the software architecture. They also gather feedback from the developers concerning the architecture, perform the coaching and have an overview about the individual skill levels of each developer. The lead developer is then responsible for passing the information about the current state of the implementation with the software architect.

Another factor may be the development process which is applied in a software project. The interview participants work in a project applying an agile, a waterfall-like process, or a mix of both. In the mixed process, the most important requirements are first assessed in a waterfall-like process, then the software development follows an agile process. Although we did not directly assess this in our study, we think that the development process may have an influence on the role of the architect, the concerns he might have and especially which enforcement activities he performs. For example, participant **H** works in a project applying a waterfall-like process. This was necessary due to strict requirements given by government regulations. As he stated, an agile process was definitely out of question due to those requirements. His main concern was the rapid deployment of changed or new components. That is why, he strives for loosely coupled components and enforced that components were appropriately partitioned according to the separation of concerns principle. Moreover, he preferred more restrictive methods like code generation. On the other hand, participant **J** worked in an agile environment. As opposed to participant **H**, participant **J** rather preferred less restrictive methods and relies on communication and coaching strategies. Those kind of topics were no concerns of participant **J**. Participant **H** was also concerned about naming conventions, how interfaces are declared and defined, if interfaces are appropriately documented or if exception handling was implemented correctly. Those are concerns related to architectural details closer to the implementation.

Although we identified relationships between several factors - such as the development process - and the tasks an architect is responsible for during enforcement, those relationships must be investigated in more detail as they were not originally the objective of our study. This is left for future work.

9.3. Limitation of the study

In the next to paragraphs, we discuss the limitation of a) the expert study and b) the literature search.

9.3.1. Limitation of the expert study

Gasson et al. proposed the criteria confirmability, dependability, internal consistency, and transferability (Gasson, 2004) in order to evaluate qualitative studies. By describing and capturing the background of all the study participants we address transferability. Confirmability is addressed by repeatedly discussing and restructuring the categories in an iterative process. In order to address dependability we followed a research process (Section 3) and described all the steps that were conducted. In terms of internal consistency the statements and the corresponding codes were cross-checked by another researcher. Similar to other qualitative studies we have a limited number of participants. However since we wanted to generate new knowledge and not to evaluate or confirm existing knowledge we find that this limited number is acceptable.

Another limitation might be that we did not consider specific factors that could influence the experts' view on enforcement concerns. For example, skills and tasks of a software architect could influence his view about what are important

concerns and activities in context of architecture enforcement.
There is the risk that each researcher might interpret the results in different ways. This risk is minimized by letting two researchers conduct the interviews independently.
Another limitation may be that the architects were not chosen randomly, but we contacted practitioners directly through our relationships.

9.3.2. Limitation of the literature search

In our literature search we did not investigate all existing databases. This involves the risk that we did not find all relevant publications. Additionally, as "architecture enforcement" is not yet a common term, we tried to use synonyms in order to paraphrase it. That is why, we might have missed some important keywords in the search query string. Consequently, not all relevant publications are contained in the resulting data set. In a next step, publications in other databases could be search and the search string could be refined.

10. Conclusion

In this paper, we presented results from an empirical study that aims for understanding the architecture enforcement in practice. We interviewed 17 experienced software architects from several companies and analyzed the interviews using methods from grounded theory. By doing this, we discovered enforcement concerns and activities. Additionally, we compared our findings with literature. The contribution of our study is a comprehensive overview about how architects' practices are discussed in state-of-the-art and how the concepts from literature are actually used in practice.
Architects have several objectives during the enforcement process which we have called enforcement concerns. We found that many of them are discussed in literature, but are mostly not directly related with with architecture enforcement. The concerns *Appropriate use of technology* or *visibility of domain concepts in code* are two examples.
We introduced two enforcement goals and related the enforcement activities to those goals. As described in the discussion, several factors and their impact on the architecture enforcement process should be investigated in more detail, e.g. how software architects differentiate between macro architecture and micro architecture decisions. Another interesting research direction would be to investigate which perspectives developers have on software architecture and how they align with the concerns of the software architects. In our study, we investigated the software architect's point of view during architecture enforcement. It would be worth to additionally investigate the developer's view by interviewing developers who are involved in enforcement of architecture decisions, e.g. in order to discover which challenges they face during this process.

11. Acknowledgments

We gratefully acknowledge and thank all the interview participants for their contributions and the productive discussions.

Appendix A. Interview Guide

In the following we present the interview guide with the main questions. As the interview was designed as a semi-structured interview, it is possible that additional, context-dependent questions were asked. Those questions are not shown here. Moreover, the order of the guiding questions differed from interview to interview.

Part I: Briefing/Participants' Background

- Please, describe a recent project you are working for or you have worked for.
- What type of application is developed/maintained in this project?
- What are the important architecture decisions that were made?
- How large is the development team and the architect team, respectively?

- How many years of experience do you have as an architect?
- Do you have other duties in the project beside software architecture related tasks? If yes, which types of duties?

Part II: Enforcement Concerns

- Regarding the most important architecture decisions in the project, which aspects and concerns of those decisions are most important and should be definitely followed by the implementation?
- What are typical/critical/recurring problems concerning those decisions (in the code)? Or more specifically which types of architectural violations do you often see in the implementation?
- In case you participate in source code reviews, on which source code aspects that are important regarding architecture do you focus especially?

Part III: Enforcement Activities

- How do you ensure that your architecture and your concerns are implemented as intended? Do you follow any strategies?
- Are you involved in implementation task or are you working close to the implementation?
- In case you participate in source code reviews, what are the specific steps you perform when you inspect the source code in order to assess the implementation of the architecture decisions?
- Do you define formal rules for architecture? What type of rules are they? Do you check them automatically?
- How do you use architecture documents and models during the enforcement process?

References

Abrahamsson, P., Babar, M.A., Kruchten, P., 2010. Agility and architecture: Can they coexist? IEEE Software 27, 16–22. doi:10.1109/MS.2010.36.

Aldrich, J., Chambers, C., Notkin, D., 2002. Archjava: Connecting software architecture to implementation, In: Proceedings of the 24th International Conference on Software Engineering, ACM, New York. pp. 187–197. doi:10.1145/581339.581365.

Antonino, P.O., Morgenstern, A., Kuhn, T., 2016. Embedded-software architects: It's not only about the software. IEEE Software 33, 56–62. doi:10.1109/MS.2016.142.

B. Kitchenham, S.C., 2007. Guidelines for performing systematic literature reviews in software engineering, In: Technical report, Ver. 2.3 EBSE Technical Report. EBSE. sn.

Babar, M.A., 2009. An exploratory study of architectural practices and challenges in using agile software development approaches, In: 2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture, pp. 81–90. doi:10.1109/WICSA.2009.5290794.

Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review, In: Proceedings of the 2013 International Conference on Software Engineering. doi:10.1109/ICSE.2013.6606617.

Baker, Jr., R.A., 1997. Code reviews enhance software quality, In: Proceedings of the 19th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 570–571. doi:10.1145/253228.253461.

Bass, L., Clements, P., Kazman, R., 2012. Software Architecture in Practice. 3rd ed., Addison-Wesley Professional, Boston.

Beck, K., 2000. Extreme Programming Explained: Embrace Change. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Berenbach, B., 2008. The other skills of the software architect, In: Proceedings of the First International Workshop on Leadership and Management in Software Architecture, ACM. pp. 7–12.

Britto, R., Smite, D., Damm, L.O., 2016. Software architects in large-scale distributed projects: An ericsson case study. IEEE Software 33, 48–55. doi:10.1109/MS.2016.146.

Buchgeher, G., Weinreich, R., 2011. Automatic tracing of decisions to architecture and implementation, In: Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture, IEEE, Washington, DC. pp. 46–55. doi:10.1109/WICSA.2011.16.

Buschmann, F., 2009. Introducing the pragmatic architect. IEEE Software 26, 10–11. doi:10.1109/MS.2009.130.

Buschmann, F., 2010. Learning from failure, part iii: On hammers and nails, and falling in love with technology and design. IEEE Software 27, 49–51. doi:10.1109/MS.2010.47.

Buschmann, F., 2011a. Gardening your architecture, part 1: Refactoring. IEEE Software 28, 92–94. doi:10.1109/MS.2011.76.

Buschmann, F., 2011b. Tests: The architect's best friend. IEEE Software 28, 7–9. doi:10.1109/MS.2011.65.

Buschmann, F., 2011c. Unusable software is useless, part 2. IEEE Software 28, 100–102. doi:10.1109/MS.2011.48.

Buschmann, F., 2011d. Unusable software is useless, part 2. IEEE Software 28, 100–102. doi:10.1109/MS.2011.48.

Buschmann, F., 2012. To boldly go where no one has gone before. IEEE Software 29, 23–25. doi:10.1109/MS.2012.18.

Buschmann, F., Henney, K., 2010. Five considerations for software architecture, part 1. IEEE Software 27, 63–65. doi:10.1109/MS.2010.72.
Caracciolo, A., Lungu, M.F., Nierstrasz, O., 2014. How do software architects specify and validate quality requirements?, In: Proceedings of the 8th European Conference of Software Architecture. Springer International Publishing, Cham, pp. 374–389. doi:10.1007/978-3-319-09970-5_32.
Charmaz, K., 2014. Constructing grounded theory. 2nd ed., Sage, London.
Christensen, H.B., Hansen, K.M., 2010. An empirical investigation of architectural prototyping. Journal of Systems and Software 83, 133 – 142. URL: http://www.sciencedirect.com/science/article/pii/S0164121209001903, doi:https://doi.org/10.1016/j.jss.2009.07.049. sI: Top Scholars.
Christensen, H.B., Hansen, K.M., Schougaard, K.R., 2009. An empirical study of software architects' concerns, In: Proceedings of the 16th Asia-Pacific Software Engineering Conference, IEEE, Washington, DC, USA. pp. 111–118.
Clerc, V., Lago, P., van Vliet, H., 2007. The Architect's Mindset. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 231–249. URL: https://doi.org/10.1007/978-3-540-77619-2_14, doi:10.1007/978-3-540-77619-2_14.
Ducasse, S., Pollet, D., 2009. Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering 35, 573–591. doi:10.1109/TSE.2009.19.
Erder, M., Pureur, P., 2016. What's the architect's role in an agile, cloud-centric world? IEEE Software 33, 30–33. doi:10.1109/MS.2016.119.
Faber, R., 2010. Architects as service providers. IEEE Softw. 27, 33–40. URL: http://dx.doi.org/10.1109/MS.2010.37, doi:10.1109/MS.2010.37.
Fagan, M.E., 1999. Design and code inspections to reduce errors in program development. IBM Syst. J. 38, 258–287. doi:10.1147/sj.382.0258.
Fairbanks, G., Garlan, D., 2010. Just Enough Software Architecture: A Risk-driven Approach. Marshall & Brainerd, Boulder.
Ford, N., Parsons, R., Kua, P., 2017. Building Evolutionary Architectures: Support Constant Change. " O'Reilly Media, Inc.".
Fowler, M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley, Boston.
Fowler, M., 2003. Who needs an architect? IEEE Software 20, 11–13.
Gall, H., Hajek, K., Jazayeri, M., 1998. Detection of logical coupling based on product release history, In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), pp. 190–198. doi:10.1109/ICSM.1998.738508.
Galster, M., Angelov, S., Meesters, M., Diebold, P., 2016. A multiple case study on the architect's role in scrum, In: Abrahamsson, P., Jedlitschka, A., Nguyen Duc, A., Felderer, M., Amasaki, S., Mikkonen, T. (Eds.), Product-Focused Software Process Improvement, Springer International Publishing, Cham. pp. 432–447.
Gasson, S., 2004. Rigor in grounded theory research: An interpretive perspective on generating theory from qualitative field studies. The handbook of information systems research , 79–102.
Glaser, B.G., 1978. Theoretical sensitivity: Advances in the methodology of grounded theory. Sociology Pr.
Gokyer, G., Cetin, S., Sener, C., Yondem, M.T., 2008. Non-functional requirements to architectural concerns: Ml and nlp at crossroads, In: 2008 The Third International Conference on Software Engineering Advances, pp. 400–406. doi:10.1109/ICSEA.2008.28.
Groene, B., Kleis, W., Boeder, J., 2010. Educating architects in industry - the sap architecture curriculum, In: 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, pp. 201–205. doi:10.1109/ECBS.2010.28.
Herold, S., Mair, M., Rausch, A., Schindler, I., 2013. Checking conformance with reference architectures: A case study, In: 2013 17th IEEE International Enterprise Distributed Object Computing Conference, pp. 71–80. doi:10.1109/EDOC.2013.17.
Hoda, R., Murugesan, L.K., 2016. Multi-level agile project management challenges: A self-organizing team perspective. Journal of Systems and Software 117, 245 – 257. URL: http://www.sciencedirect.com/science/article/pii/S0164121216000807, doi:https://doi.org/10.1016/j.jss.2016.02.049.
Hove, S.E., Anda, B., 2005. Experiences from conducting semi-structured interviews in empirical software engineering research, In: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE, Washington, DC. pp. 10–23.
ISO, 2011. Road vehicles – Functional safety.
Jansen, A., Bosch, J., 2005. Software architecture as a set of architectural design decisions, In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, IEEE, Washington, DC. pp. 109–120.
Klein, J., 2005. How does the architect's role change as the software ages?, In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), pp. 141–141. doi:10.1109/WICSA.2005.38.
Klein, J., 2016. What makes an architect successful? IEEE Software 33, 20–22. doi:doi.ieeecomputersociety.org/10.1109/MS.2016.9.
Knöpfel, A., Gröne, B., Tabeling, P., 2005. Fundamental modeling concepts:Effective Communication of IT Systems. Wiley, England.
Kollanus, S., Koskinen, J., 2009. Survey of software inspection research. The Open Software Engineering Journal 3, 15–34.
Kruchten, P., 1999. The Software Architect. Springer US, Boston, MA. pp. 565–583. URL: https://doi.org/10.1007/978-0-387-35563-4_33, doi:10.1007/978-0-387-35563-4_33.
Kruchten, P., 2000. The Rational Unified Process: An Introduction, Second Edition. 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Kruchten, P., 2008. What do software architects really do? Journal of Systems and Software 81, 2413 – 2416.
Lagerstedt, R., 2014. Using automated tests for communicating and verifying non-functional requirements, In: 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), pp. 26–28. doi:10.1109/RET.2014.6908675.
Lungu, M., Lanza, M., Nierstrasz, O., 2014. Evolutionary and collaborative software architecture recovery with softwarenaut. Science of Computer Programming 79, 204 – 223. doi:http://dx.doi.org/10.1016/j.scico.2012.04.007. experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).
Martini, A., Pareto, L., Bosch, J., 2014. Role of Architects in Agile Organizations. Springer International Publishing, Cham. pp. 39–50. URL: https://doi.org/10.1007/978-3-319-11283-1_4, doi:10.1007/978-3-319-11283-1_4.
McBride, M.R., 2004. The software architect: Essence, intuition, and guiding principles, In: Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA. pp. 230–235. URL: http://doi.acm.org/10.1145/1028664.1028764, doi:10.1145/1028664.1028764.
McBride, M.R., 2007. The software architect. Commun. ACM 50, 75–81.

Mirakhorli, M., Cleland-Huang, J., 2016. Detecting, tracing, and monitoring architectural tactics in code. IEEE Transactions on Software Engineering 42, 205–220. doi:10.1109/TSE.2015.2479217.

Murphy, G.C., Notkin, D., 1997. Reengineering with reflexion models: a case study. Computer 30, 29–36. doi:10.1109/2.607045.

Pareto, L., Eriksson, P., Ehnebom, S., 2012. Concern coverage in base station development: an empirical investigation. Software & Systems Modeling 11, 409–429. URL: https://doi.org/10.1007/s10270-010-0188-2, doi:10.1007/s10270-010-0188-2.

Paulisch, F., Zimmerer, P., 2010. A role-based qualification and certification program for software architects: an experience report from siemens, In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, pp. 21–27. doi:10.1145/1810295.1810300.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17, 40–52.

Pruijt, L.J., Köppe, C., van der Werf, J.M., Brinkkemper, S., 2014. Husacct: Architecture compliance checking with rich sets of module and rule types, In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, New York. pp. 851–854. doi:10.1145/2642937.2648624.

Rumbaugh, J., Jacobson, I., Booch, G., 2004. Unified Modeling Language Reference Manual, The (2Nd Edition). Pearson Higher Education.

Sauer, J., 2010. Architecture-Centric Development in Globally Distributed Projects. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 321–329. URL: https://doi.org/10.1007/978-3-642-12442-6_22, doi:10.1007/978-3-642-12442-6_22.

Schröder, S., Riebisch, M., Soliman, M., 2016. Architecture enforcement concerns and activities-an expert study, In: Proceedings of the 10th European Conference on Software Architecture (ECSA2016), Springer. pp. 247–262.

Schwaber, K., Beedle, M., 2001. Agile Software Development with Scrum. 1st ed., Prentice Hall PTR, Upper Saddle River, NJ, USA.

Sherman, S., Hadar, I., 2015. Toward defining the role of the software architect: An examination of the soft aspects of this role, In: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 71–76. URL: http://dl.acm.org/citation.cfm?id=2819321.2819333.

Sherman, S., Unkelos-Shpigel, N., 2014. What do software architects think they (should) do?, In: Iliadis, L., Papazoglou, M., Pohl, K. (Eds.), Advanced Information Systems Engineering Workshops, Springer International Publishing, Cham. pp. 219–225.

Staron, M., Meding, W., 2017. A portfolio of internal quality metrics for software architects, In: Winkler, D., Biffl, S., Bergsmann, J. (Eds.), Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies, Springer International Publishing, Cham. pp. 57–69.

Strauss, A., Corbin, J., et al., 1990. Basics of qualitative research. volume 15. Sage, Newbury Park, CA.

Tamburri, D.A., Kazman, R., Fahimi, H., 2016. The architect's role in community shepherding. IEEE Software 33, 70–79. doi:10.1109/MS.2016.144.

Taylor, R.N., Medvidovic, N., Dashofy, E.M., 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

Terra, R., Valente, M.T., 2009. A dependency constraint language to manage object-oriented software architectures. Software: Practice and Experience 39, 1073–1094. doi:10.1002/spe.931.

Thomas, S.W., Hassan, A.E., Blostein, D., 2014. Mining unstructured software repositories, In: Evolving Software Systems. Springer, pp. 139–162.

Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., i. Matsumoto, K., 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review, In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 141–150. doi:10.1109/SANER.2015.7081824.

Unphon, H., Dittrich, Y., 2010. Software architecture awareness in long-term software product evolution. Journal of Systems and Software 83, 2211 – 2226. URL: http://www.sciencedirect.com/science/article/pii/S0164121210001743, doi:https://doi.org/10.1016/j.jss.2010.06.043. interplay between Usability Evaluation and Software Development.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering, In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, ACM, New York, NY, USA. pp. 38:1–38:10. URL: http://doi.acm.org/10.1145/2601248.2601268, doi:10.1145/2601248.2601268.

Woods, E., 2015. Aligning architecture work with agile teams. IEEE Software 32, 24–26. doi:10.1109/MS.2015.119.

Zimmermann, O., 2009. An architectural decision modeling framework for service oriented architecture design. Ph.D. thesis. URL: http://dx.doi.org/10.18419/opus-2665, doi:10.18419/opus-2665.

Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., Schuster, N., 2007. Reusable architectural decision models for enterprise application development, In: Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications, Springer, Berlin. pp. 15–32.