Supplementary Material: Architecture Enforcement Concerns and Activities - An Expert Study

Sandra Schröder, Matthias Riebisch, and Mohamed Soliman

University of Hamburg, Department of Informatics,
Vogt-Koelln-Strasse 30, 22527 Hamburg, Germany

1 Enforcement Concerns

As Enforcement Concerns (Fig. 1) we summarized all aspects that have to be assessed by architects to ensure the correct implementation of decisions, which may include to validate if an implementation does not violate the architecture. These assessments may be performed on all types of artifacts that are part of an implementation, such as source code, frameworks, building blocks or third party products. Table 1 gives an overview over the identified concerns from the interviews.

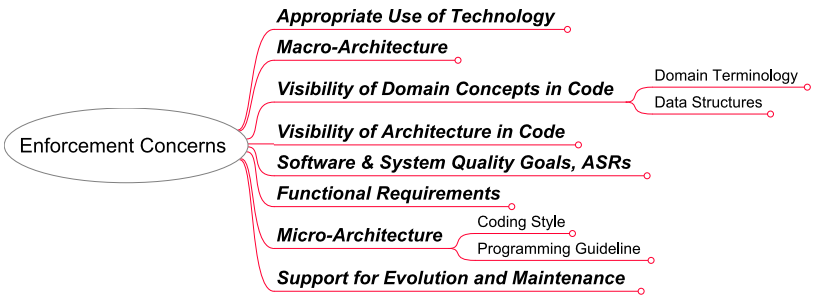

Fig. 1. Overview of Enforcement Concerns in a mindmap. The full mindmap with attached codes is available in the supplementary on the paper's website[3].

1.1 Architecture Design Principles

The results confirm the importance of ensuring architectural design principles. As can be seen in Table 1 most of the participants mentioned architectural design

Table 1. Overview of the identified categories of Enforcement Concerns from the interviews. Concerns marked with an asterisk are not explained in detail in this paper but available in the supplementary on the paper's website[3].

Enforcement Concern	**Participants**
Macro and Micro Architecture Decisions	**B**, **C**, **D**, **H**, **I**, **J**, **K**
Architecture Design Principles	**A**, **C**, **D**, **E**, **F**, **G**, **H**, **J**, **K**, **L**
Appropriate Use of Technology	**A**, **D**, **H**, **J**, **L**
Patterns	**C**, **I**, **J**, **K**
Visibility of Domain Concepts in Code	**C**, **D**, **E**, **J**, **K**
Visibility of Architecture in Code	**A**, **C**, **E**, **J**
Software Quality and ASRs	**A**, **G**, **I**
Functional Requirements	**I**, **J**, **L**
Code Comprehensibility	**D**, **J**, **K**
Design for Testability	**D**, **J**

principles as important concerns. For example, participant L aimed a modular structured system composed of loose coupled components: *"[the system] is composed of very loose coupled modules that only communicate via asynchronous messages [...] this is also very important architecture principle. No coupling..."* (codes: loose coupling, modularization; Participant L). Dependencies were also mentioned as an important concern. Experts strive to have only a few dependencies between modules in order to avoid a complex structure and additionally to avoid a high coupling between modules (principle loose coupling). Cyclic dependencies between two software elements have long been recognized as an indicator of tight coupled systems. If any cycles in a system exist, functionalities cannot be understood and tested separately. Moreover changes on functionalities can only be implemented with great effort. That is why experts also validate if the implementation contains any cyclic dependencies between components in order to ensure loose coupling. For example, participant **H** stated that developers tend to create monolithic structures that are, consequently, hard to maintain and to deploy. That is why he strictly enforces that software is well-structured in terms of loose coupled components with only few dependencies. Experts mentioned a lot of other principles; due to space limitation we cannot present all of them and we refer to the supplementary material on the paper's website for interested reader. Table 2 shows an excerpt of statements made by participants concerning architectural principles such as separation of concerns, modularization or minimizing dependencies.

1.2 Software Quality and ASRs

Ensuring software quality and architectural significant requirements (ASRs) are also mentioned as important concerns for enforcement. For example participant A stated that quality attributes such as security, scalability and performance

were of great importance in a recent project. That is why he especially focused on pieces of codes that could potentially violate decisions that refer to specific ASRs and quality attributes: *"...security architecture is important. For example http. If parts from the domain model such as orders or credit card numbers are stored in a cookie then this is an architectural violation obviously concerning the decisions "We use data-based session state or server session state and not client session state." This has a high priority. This has to be repaired immediately. If I find something like this in the code I would stop the operation of the system immediately. [...] This can be detected automatically, by checking if the cookie API is used somewhere..."* (code: violation - security violating code structure; participant A).

1.3 Functional Requirements

Besides quality attributes and ASRs functional requirement also play a significant role during enforcement. Nevertheless we skipped its detailed description in the main paper. Functional requirements also define types of constraints that have to be ensured and enforce during implementation and maintenance. For example participant I stated that: *"...are the agents able to communicate with arbitrary types of databases and to extract data and to forward these data to other agents. This would be a requirement that is included in the validation."* (code: functional requirements as constraints, participant I).

1.4 Macro and Micro Architecture Decisions.

When talking about software architecture, it was interesting that experts differentiate basically between decisions in two different views, namely macro architecture and micro architecture [1]. Other terms like strategic or global (i.e. macro) and tactical or local (i.e. micro) views were used. The architect can decide which decisions are located in the macro architecture, and which decisions are left open for the development team. In this way the architect can decide how much freedom he gives teams in designing the micro architecture. The macro architecture represents the general idea (or "philosophy", the "spirit", the "big picture" or metaphor) of the system and its fundamental and most critical architectural decisions, e.g. on structures, components, data stores, communication style or architectural styles: *"...it is important how you regard it. For me there do exist 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) and another participant reported: *"...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). Those two views define what architects basically consider as important for architecture enforcement in different ways. Architects reported to be concerned with macro architecture issues and consider the micro architecture as

developers' responsibility, except the coding style because of its relevance for maintainability: *"...architecture is also present in a single code statement. Code styles belong to it. Or simple things like how do I define an interface..."* (code: micro architecture, Participant J).
This distinction is also described in [1]. There macro architecture is described as *"the spectrum of the architecture levels to which architecturally relevant elements are assigned (organizational level and system level, as well as the part of the building block level on which fundamental system building blocks"*, whereas the authors define micro architecture as *"the detailed design (small-scale architecture) closely associated with the source code with no fundamental influence on an architecture"*. The distinction between those two views is also known in the context of micro service architectures [**?**]. There the macro architecture describes architecture decisions that are important for the whole system, that is decisions concerning the functional separation into single services, the basic technologies, communication and integration styles or security decisions, whereas the micro architecture contains decisions that the team can decide for itself for a specific micro service.

1.5 Appropriate Use of Technology

was also mentioned as an important concern. The architect may not check technologies used within a single component, but may for example enforce the technology for the communication style between several components. Since technologies like frameworks or libraries offer a lot of complex functionality, software architects also monitor the way how those technologies are used by developers. One architect stated that developers can easily violate important architecture rules due to this complexity. Some architects reported 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 gross things, but those gross things cannot be controlled..."* (code: aim for technologies, Participant J). They stated that software architecture is likely to erode where too much technology is used, because this part of code is hard too understand (see also "Support for Evolution and Maintenance"). This is why some experts emphasized it is important to control what kind of technologies are used.

1.6 Patterns

The architect may want to ensure that the important patterns are implemented accordingly. Patterns related with the macro architecture view have to be enforced and validated, patterns on the micro level are mostly considered as a developers' concern. But sometimes, pattern implementations are also checked by architects on the micro architecture level, e.g. in order to discover what types of design and architecture patterns are implemented and if they fit in the specific context: *"which 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 layered pattern seems to be a predominant pattern and is considered as an important aspect by most of the interviewed experts (**A**, **E**, **F**, **G**, **J**, **K**, **L**). Participant J relates the layer pattern to the macro architecture level. The layer pattern can be violated easily - as stated by software architects - that means that lower layers use functionality of upper layers. Other well-known patterns are also checked against violations, e.g. the MVC pattern, for example to check if the view component does not contain any business logic code: *"...what happens sometimes is that code is put in patterns or components that does not belong to it, this happens again and again..."* (code: pattern unrelated code, Participant K). Some software architects also check if other architectural or design patterns are implemented appropriately, that is if the pattern code does not contain non-pattern related code in pattern realizations or if changes broke the structural or functional integrity of a pattern: *"...what happens sometimes is that code is put in patterns or components that does not belong to it, this happens again and again..."* (code: pattern unrelated code, Participant K).

1.7 Visibility of Domain Concepts in Code

Some experts emphasize a clear representation of domain concepts in the architecture, e.g. by expressing a mapping between them. For this, some architect strive to use a domain oriented terminology, that means using terms and names adapted from the business 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 additionally helps to talk with domain experts about the software design and to easier locate where changes have to be made in case of new or adapted requirements.

1.8 Visibility of Architecture in Code

Some architects consider it as important to make the architecture visible in the code, e.g. by using appropriate naming conventions and package structure: *"...therefore it is important that the architecture is recognizable in the source code. This is absolutely essential for the structure of the project."* (code: making architecture visible in the code, Participant J). This is helpful for tools like Sonargraph[1] that for example use naming conventions in order to highlight layers. It was also mentioned to be useful during code inspections in order to easily locate architecture decisions in code. This concern is similar to the idea using an architecturally-evident coding style suggested by Fairbanks [2].

[1] https://www.hello2morrow.com/products/sonargraph

1.9 Support for Evolution and Maintenance

A challenge in constructing long-lived system is to make decisions that support the software system's ability to easily be adapted to future changes, that is we need support for evolution and maintenance during the entire software lifecycle.

- **Code Comprehensibility** was explicitly mentioned as a concern, on the basis that comprehensibility helps preventing architectural violations: *"if you strictly follow this approach then you have very readable code and normally readable Code - from my experience - tends to be stable that is conform concerning architecture and does not have any [architecture] violations..."* (code: code comprehensibility, code comprehensibility supports architecture conformance; Participant J).
- **Terminology** was reported being important for code comprehensibility: *"...terminology is extremely important for comprehension and if the comprehension is good, the code is more readable"* (code: terminology enables comprehensibility, Participant J). If the terminology is chosen appropriately it is easier to understand what a specific piece of code is about: *"...is it named meaningful? This is one of the most important things, the naming of classes. From the name you can derive what [the class] does..."* (code: terminology, Participant K).
- **Design for Testability** Another interesting aspect that might be surprising was that architects are strongly concerned with tests. Systems that cannot be properly tested, cannot be changed successfully since software modifications during maintenance and implementation activities may lead to errors. That is why testability is an important concern especially in the context of evolution and maintenance. Participants aim a high test coverage in order to avoid architectural violations: *"...in case there exist only a few tests, then it is likely people do not build it correctly. This leads to incomprehensible code and consequently to architectural violations."* (test coverage supports architecture conformance, Participant J). Tests are therefore an important concern for enforcement. Moreover, technologies can prevent testability. For example participant L forbids using queues and only allowed topics in JMS to be used in order to ensure testability.

References

1. O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software architecture: a comprehensive framework and guide for practitioners.* Springer Science & Business Media, 2011.
2. G. Fairbanks, *Just enough software architecture: a risk-driven approach.* Marshall & Brainerd, 2010.

Table 2. Excerpt of the concept "Architectural Principles" emerged from open codes and example statements made by participants

Open Code	**Incident**	**Participant**
Separation of Concerns	*"and if the component is not properly mapped onto a category then it is mostly mapped onto several categories and this implies that the layering is not followed [...] and if you violate it then [...] you have to start a separate project in order to recover this separation."*	**E**
Modularization	*"[checked] that the developers build modules, they implemented the interfaces of the modules correctly, that the encapsulation was correct..."*	**L**
	"how modular is the system. We thought in terms of modules, we visualized building blocks in the structural view. But how does it really look like at the building blocks' boundaries? Is there a strong interweaving with the environment, so that we cannot reuse the building blocks as we intended first? ..."	**G**
Minimizing Dependencies	*"... so that you only have a small depth in the dependency tree, maybe 4 or 5. I also saw 10 but it is then impossible to understand the relations..."*	**J**
	"that a component cannot be separated, since it has more dependencies and unplanned dependencies as we thought..."	**C**
	"the result were dependencies that did not belong in this part. A monolithic application. This is the worst that can happen."	**H**
Layering	*..."when we decided for a layer architecture we took care that the layering is ensured..."*	**H**
	..."that I only have the defined relations between the layers"	**J**
Loose Coupling	*"...this is the architecture of the controlling system. It is composed of loosely coupled modules that only communicate over asynchronous messages. This appeared to be very beneficial."*	**K**
	"...in the context of enterprise application you always try to achieve loose coupling between components..."	**D**