# T U M

I N S T I T U T F Ü R I N F O R M A T I K

# Designing, Documenting, and Evaluating Software Architecture

David Bettencourt da Cruz, Birgit Penzenstadler

TUM-I0818
Juni 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I0818-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2008

Druck: Institut für Informatik der
Technischen Universität München

# Designing, Documenting, and Evaluating Software Architecture

David Bettencourt da Cruz    Birgit Penzenstadler

June 25, 2008

**Abstract**

Part of any engineering discipline is the use of systematic, repeatable and traceable processes, methods and procedures. Software architecture is a branch of the discipline of software engineering. However, software architecture design, documentation and evaluation is still lacking a sound basis of systematic, repeatable and traceable procedures. At present software architecture is still more of an art open only to skilled and experienced experts rather than an engineering discipline open to the majority of software engineers.

As software systems become more and more complex the importance of software architecture throughout the whole development cycle increases. Software architecture is not only a means to tackle complexity of large software systems but also a means to enable system qualities. Hence, it is becoming more and more important to include software architecture engineering into the software engineering discipline and the software development process.

In this report we present systematic, repeatable and traceable methods of developing, documenting and evaluating software architectures in order to provide a basis for further research.

**Keywords:** ATAM, POSAAM, POSA, Architecture Analysis, Architecture Evaluation, Architecture Documentation, Architecture Construction, Software Patterns.

**Acknowledgement:** We would like to thank Gerd Beneken for his valuable comments and feedback.

# Contents


**1 Introduction** **3**

**2 Running example: Navigation system** **5**
- 2.1 Business Needs . . . 6
- 2.2 Requirements Specification . . . 6
- 2.3 System Specification . . . 10
- 2.4 Project Plan and Management . . . 10

**3 Designing an Architecture** **11**
- 3.1 Directive Criteria . . . 11
- 3.2 Functional Criteria . . . 12
- 3.3 Quality Criteria . . . 13
- 3.4 Architectural Criteria . . . 14
- 3.5 Decomposing the system according to the criteria . . . 14
- 3.6 Further guidance . . . 16

**4 Documentation** **18**
- 4.1 Shortcomings of Architecture Documentation . . . 18
- 4.2 Documenting Layers, Views, and Rationale . . . 19

**5 Shortcomings of Common Architecture Analysis Methods** **24**
- 5.1 Analysis of the Systematic Guidance for Architecture Evaluation in ATAM . . . 24
- 5.2 Goals for a New Qualitative Architecture Evaluation Method . . . 26

**6 Evaluating an Architecture Using POSAAM** **27**
- 6.1 Moving from Requirements to Patterns . . . 28
- 6.2 Networking: Using Pattern Relations . . . 28
- 6.3 Using Pattern Knowledge for Evaluation . . . 29
- 6.4 Special Issues . . . 30
  - 6.4.1 Evaluating Using Principles . . . 30
  - 6.4.2 Structuring Expertise Stored in Patterns . . . 32

**7 Related Work** **35**

**8 Conclusion** **37**
- 8.1 How the goals of POSAAM have been addressed . . . 37
- 8.2 Further work . . . 38

**References** **43**

# 1 Introduction

Besides the fulfilment of functional requirements, quality requirements have become an important factor to the success of software products. It is generally recognized that the software architecture has an influence on the quality attributes of a system [3]. However, the task of systematically designing software architectures has not been discussed thoroughly. Furthermore, the role of software architecture is not limited to early blueprints and the organization of the division of labour during development, but also serves as basis for understanding a system and later on integrating and evolving it. This wide-stretched notion of architecture requires not only to analyse and explain the design, but also the documentation of an architecture. Finally, when laying an emphasis on the quality attributes of a system, the continuous analytical and constructive quality assurance during the development process is an obligatory practice. One of the earliest products in software development and at the same time one of the most influencing for the overall quality of the development process is the software architecture specification – the architect's vision of the system to be developed. The quality of the resulting system can and should already be assessed at this early stage. At the latest, before starting with the implementation of the system. As architecture evaluation happens early in development, it can be a cost effective way to discover misconceptions which could lead to decreased quality or even costly reengineering of the final software product.

In this report, we detail how to design, document, and evaluate an architecture, illustrated with the example of a navigation system.

We show which criteria other than the quality attributes have an influence on the software architecture to be developed and how these criteria should be addressed within the design stage. For this purpose we give an explanation of the criteria, which are categorized into directive, functional, quality and architectural criteria and demonstrate how these should be used to lead to the systems architectural decomposition.

In software architecture documentation the concept of using views is widely accepted (e.g. [26, 15]). However, the design of software architecture is largely a process of considering alternative possibilities and explicitly taking the decision for a chosen alternative for specific reasons. The software architecture documentation should reflect this process because the decisions taken should be traceable for development and later evolution and maintenance processes. Our proposal on how to document software architectures consist of a combination of using views and of using decision templates as proposed by Tyree et al. [47].

In [16] Clements et al. correctly state that if it is possible to influence a system's quality attributes through the system's architecture, then it must be possible to determine whether a given architecture has the influence on quality attributes which is desired. However, the methods available for evaluating software architectures do not focus on this aspect of evaluation. Rather do they set their focus on correctly eliciting requirements and on different aspects of social engineering during the evaluation process. Furthermore, it is only natural to discover that the specified architecture does not address the stakeholders

requirements if these were not known to the architect in the first place.

We believe that architecture evaluation should answer the question: "Did the architect consider the accepted well-known techniques used to achieve the required qualities? And if not, why?". To answer this question we analyse what the accepted techniques to achieve qualities are, i.e. how an architecture should be designed in the first place, and how this knowledge can be used to perform an architecture evaluation.

Within our architecture evaluation method we assume that the requirements have already been elicited and that an architectural description is available. While the description of quality requirements as scenarios (as defined by [3]), which is also used in other common architecture evaluation methods [16], is perfectly useful for us, the description of software architecture we require needs to be specified before our method can be presented. Therefore we address the description of software architecture just before presenting our evaluation method.

**Contribution:** We explain our understanding of how to construct and document architectures. Based on that, we discuss the shortcomings of common architecture evaluation methods and present a method for architecture analysis that offers a solution for the described insufficiencies.

**Outline:** For demonstration purposes we introduce a fictitious example in Section 2, which we will refer to throughout the remaining sections. We explain how to include the different influences on the architecture during design in Section 3 and how we require architecture to be documented for our evaluation method in Section 4. In Section 5 we present the shortcomings of traditional software architecture evaluation methods and derive goals for our own evaluation method POSAAM which is detailed in Section 6. Finally, Section 7 relates to other work and Section 8 draws conclusions, presents how the goals for evaluation methods have been addressed by POSAAM and identify starting points for further work.

## 2 Running example: Navigation system

The running example we will use throughout the paper is a car navigation system, which we will refer to as NAVI. This section describes its requirements, setting, and constraints. It is a fictitious system of a navigation system featuring normal address routing and point-of-interest search.

We have structured the presentation of NAVI according to the Requirements Engineering and Management model (REM) developed by Geisberger et al [23]. Within REM there are three content categories for specification development artefacts: Business Needs, Requirements Specification, and System Specification — as can be seen in Fig. 1. Apart from the three content categories, there is the orthogonal aspect of project planning and management that captures the process requirements and constraints.

The arrows from left to right indicate the refinement relationships between the business needs, the requirements specification and the system specification. This does not imply that the process of developing and refining the requirements is waterfall-like. The feedback loops contained in REM have been omitted in the figure for simplicity reasons. The arrows from the content categories to the project management indicate dependency on the process.

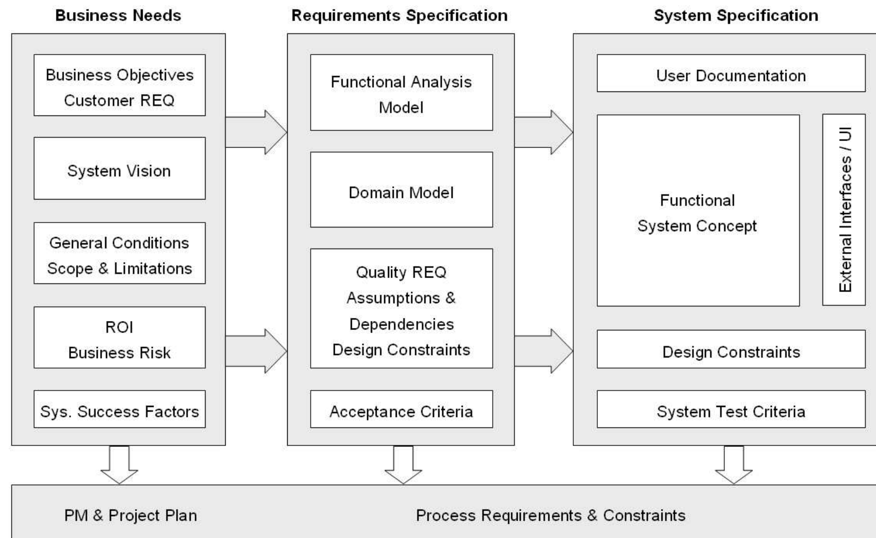


Figure 1: Artefacts of the Requirements Engineering and Management (REM) Model

In the following we will explain REM only as far as necessary to present our example, NAVI. For detailed information about REM the reader should refer to [23].

## 2.1 Business Needs

The Business Needs capture business objectives and customer requirements, the system vision, general conditions and constraints, business risks and ROI, and system success factors. To completely capture the relevant business needs, it is important to consider all stakeholders, e.g. customers, managers, developers, hardware suppliers, and service personnel. For the navigation system, some examples are:

- Business objectives:
  - Optimize service for customers.
  - Operate on different platforms to be independent of supplier.
- Customer requirement: Easy to handle, comfortable navigation system.
- System vision: Driver assistance system with navigation to geographic destinations and points-of-interest (POI).
- General conditions:
  - Usability: "Handling without looking" according to road traffic regulations (in Germany: StVO).
  - HMI: Given human machine interface (graphical display / input button).
  - Communication: Requires satellite connection.
- Business risks and ROI: adequate marketing to ensure that the customer realizes the advantages of this particular navigation system.
- System success factors: be first on the market with the privacy feature in combination with navigation.

## 2.2 Requirements Specification

The Requirements Specification contains a functional analysis model, a domain model, quality requirements (including assumptions, dependencies and design constraints), and acceptance criteria. The functional analysis model contains a description of functional requirements. For illustration purposes we present a short list of functional requirements including brief descriptions in the following. Graphical representations of the functional requirements may also be included as the use cases depicted in Fig. 2.

#### Functional requirements

F1 *Navigation:* The main functionality of the system. Directs the user from the current location to the chosen destination.

F1.1 Choose destination

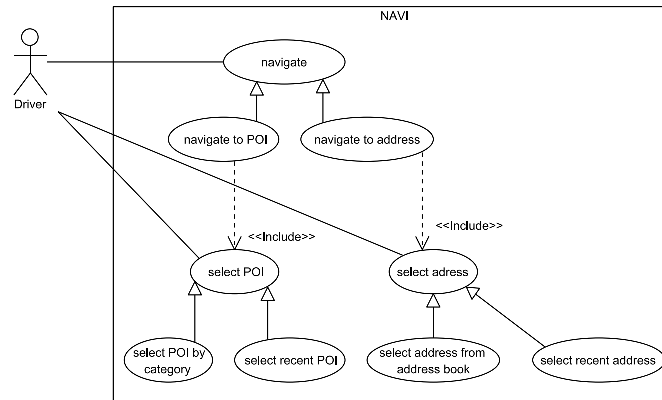


Figure 2: Use cases of the NAVI System

- F1.2 List recent destinations
- F1.3 Choose route
- F1.4 List recent routes
- F1.5 Provide alternative route
- F1.6 Toggle view (2D / 3D)
- F1.7 Expected time of arrival
- F1.8 Show current speed
- F1.9 Distance to destination
- F1.10 Time to destination
- F1.11 Adapt view to current speed

F2 *Points-of-interest:* The points-of-interest are locations of interest of a category (e.g. hospital, restaurant, hotel, recreation, ...)

- F2.1 Show next POI from current location
- F2.2 Distance to next POI
- F2.3 Duration to next POI
- F2.4 Update of POIs through internet
- F2.5 Manual update of POIs

F3 *Map:* Shows the current location in the map, which is updated as the system changes its location.

F4 *Address administration:* Provides functions to administer addresses.

- F4.1 Add address
- F4.2 Update address

F4.3 Delete address

F5 *Route planing:* Is used to plan waypoints of a route. May be stored and loaded in order to guide navigation.

F6 *Recording:* Records a taken route which may be loaded by the system and used for navigation.

F7 *Driver's logbook:* Can be used to automatically register any movement of the vehicle.

F8 *Customize:* Functions used to adapt the system to the users preferences.

F8.1 Display street names in GUI

F8.2 Voice configuration (loudness level, male/female voice, ...)

F8.3 Display colours

F8.4 Face north / face driving direction

F8.5 Speed profile

F8.6 Shortcuts

F8.7 Unit representation (configuration of representation of date, time, and distance)

F8.8 Automatically enable driver's logbook

F9 *Administrate:* Provides the possibility to perform administration tasks.

F9.1 GPS hardware configuration

F9.2 Which maps to use

F9.3 Map update

**Domain model:** Fig. 3 shows the environment of the system with all users and interacting systems. The purpose of a domain model is to give all stakeholders an intuitive overview over the system context. With NAVI, we have the advantage that everybody knows more or less exactly what this kind of system is about and what other systems are interacting in the environment of the system. Still the stakeholders involved in system development tend to make assumptions about a system's surroundings, the more expertise stakeholders have in a certain application domain, the more assumptions they make. This complicates the communication with other stakeholders who have less knowledge about the application domain. So the domain model serves mainly to give all stakeholders the same rough scope of the system's environment. The domain model shown in Fig. 3 is modelled from the perspective of the system to be developed (NAVI) at one point in time. Analogous to the modelling of systems, other models are also possible and for some applications also necessary.

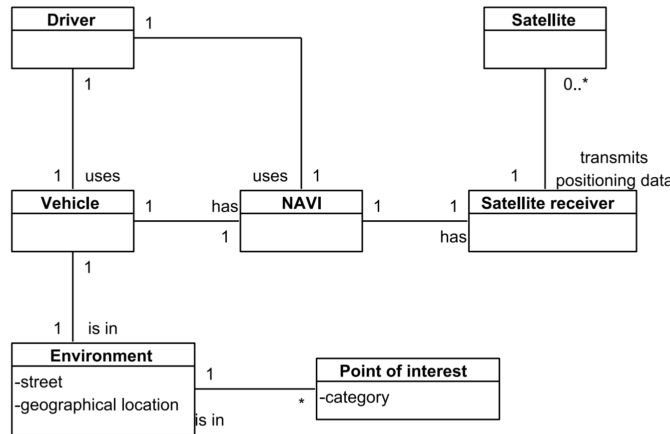


Figure 3: A simplyfied domain model for the NAVI system

**Quality requirements:** Quality requirements concern properties such as robustness ("How does the system react when it loses the satellite connection?") or efficiency (available resources, performance, electricity consumption, ...). For further demonstration purposes we will define a security quality requirement for NAVI. The customization, recent location or POI functions will be required to be user specific and assumed to contain sensible data. Thus, leading to the privacy requirement that these data only be available to the user who produced them or introduced them into the system. This means that a different driver would not be able to see the recent destinations of other drivers. Fig. 4 illustrates a use case demonstrating this requirement.

To be able to use this example in following sections, it is necessary to express this quality requirement as a scenario as defined by [3]. We define the following scenario which we will refer to as Q1:

- *Source of stimulus*: Driver
- *Stimulus*: Display information about recent destinations
- *Artefact*: The list of recent destinations
- *Environment*: The list of recent destinations contains destinations inserted by a different driver
- *Response*: The system displays recent information of the current driver and does not display information about the recent destinations of any other drivers
- *Response Measure*: The response is true.

**Acceptance criteria:** These are derived from the system success factors. The success factor "be first on the market" can hardly be transformed into an acceptance criterion, but the "privacy feature" can be an acceptance criterion. The acceptance criteria will subsequently serve as input for the derivation of test cases within the System Specification.

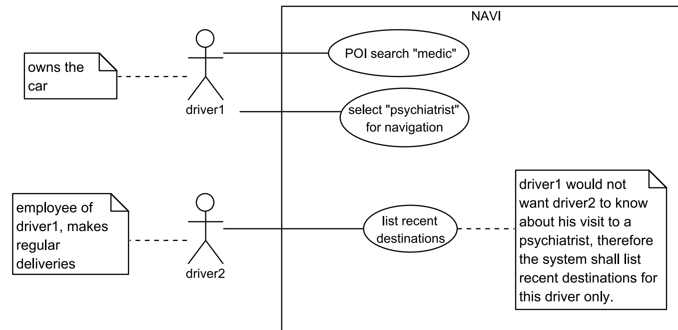


Figure 4: Use case that illustrates the privacy problem of the navigation system

## 2.3 System Specification

The System Specification includes the functional system concept (first blueprint of the design), external interfaces and UI, user documentation, design constraints and system test criteria. The functional system concept contains different views onto the system — a structural view, a behavioural view and a data view. These three also define the interface.

The System Specification and how to develop it will be explained in detail in Sec. 3.

## 2.4 Project Plan and Management

The Project Plan and the Project Management are based on an artefact-oriented process with milestones and quality gates. They are based on levels of completion of the artefacts. The development of the artefacts is processed in iterative modelling cycles that consist of the activities analysis, refinement, classification, and modelling.

# 3 Designing an Architecture

There are a lot of influences on and requirements that have to be taken into account when designing a system's architecture. When reviewing them according to their sources and impact, we distinguish four general categories: Directive Criteria, Functional Criteria, Quality Criteria, and Architectural Criteria. The categories and their correlation are depicted in Figure 5 and each of the categories is described in the following (see Sec. 3.1, 3.2, 3.3, 3.4) including the sources where to find the respective information in the REM artefact model (see Fig. 1) and illustrating them with examples from the navigation system introduced in Sec. 2.

Subsequently, Sec. 3.5 describes the process for the decomposition and the dependencies in between the categories and Sec. 3.6 gives hints for further guidance.

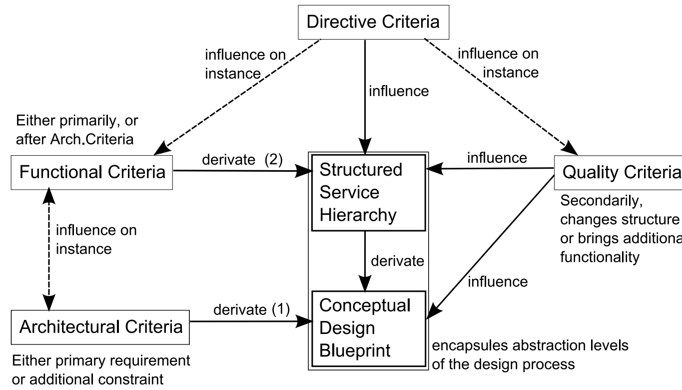


Figure 5: Influences on the Architecture

## 3.1 Directive Criteria

The directive criteria comprise management issues and constraints from the "business world". They arise from organization, legislature, and economics. The difficulty concerning the directive criteria is that in practise, they are not always documented sufficiently or even consciously present in the minds of the developers.

- **Organizational issues** are for example the company's infrastructure (confirm Conway's Law [18] and derivations [25]), supplier relationships and business rules as well as design or programming standards. Another important aspect are social issues like internal communication structures (see also [17]) although the often go unaware. Additionally, there are some company-specific heuristics that reflect best practice experiences, either the company's own experiences or best practice in their special business domain.

- **Legislative influences** may e.g. come from the data protection act, the road traffic act (see example below), or certified standards.
- **Economic influences** come from the Marketing and Sales department. They have an advertising strategy for certain features of the product, so they add their analyses and calculations for the features and set the budget constraints.

As is already visible in the picture, the influence of the directive criteria on the architecture is mainly indirect over functional and quality criteria. In other cases it may be hardly perceptible at all, e.g. when influenced by social factors.

**Information sources:** Within the REM model, some of the organizational issues will be reflected in the General Conditions and in the Business Objectives. It is probable that the company's business culture itself (social aspects and communication structures) are not documented explicitly. Legislative issues are also documented within the General Conditions and Limitations. Economic influences are captured in the ROI and Business Risk artefacts.

**Example:** An example for directive constraints from the navigation system for legislative influences is adequate usability ("handling without looking") according to road traffic regulations (in Germany: StVO). Concerning the question whether and how this would influence the architecture, the answer is that the impact is only indirect in this case via derived functional and quality criteria, but it is however important to be aware of the relations between the criteria and to give some rationale for the derived functional and quality requirements.

## 3.2 Functional Criteria

The functional criteria are the functional requirements and interaction and dependencies in between functionalities. According to [3], functionality is largely independent of structure. We perceive this as true for only certain types of functionality but not in general. A counter example is any kind of functionality that is strongly dependent on and determined by a certain quality characteristic like, e.g., security or performance. However, it is advisable to decompose the system according to the characteristic functions to enable cooperative and parallel development and to increase reuse potential.

**Information sources:** The REM model provides the information for the functional criteria in form of the System Vision, the Domain Model and the Functional Analysis Model, probably in the notation form of use cases and scenarios.

**Example:** For the navigation system, examples for functional requirements are *standard navigation* and *points-of-interest* as listed and detailed in Sec. 2.2.

### 3.3 Quality Criteria

The quality criteria are the system quality requirements defining the constraints with respect to the characteristics and subcharacteristics defined in ISO 9126 [27]:

- Functionality: A set of attributes that bear on the existence of a set of functionalities. These are Suitability, Accuracy, Interoperability, Compliance, and Security.
- Reliability: A set of attributes that bear on the capability of the system to maintain its level of performance under stated conditions for a certain period of time. These are Maturity, Recoverability, and Fault Tolerance.
- Usability: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. These are Learnability, Understandability, and Operability.
- Efficiency: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. These are Time Behaviour and Resource Behaviour.
- Maintainability: A set of attributes that bear on the effort needed to make specified modifications. These are Stability, Analyzability, Changeability, and Testability.
- Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another. These are Installability, Replaceability, and Adaptability.

These quality characteristics contain both architectural and nonarchitectural aspects, for example performance depends on an adequate component allocation (architectural) as well as on the choice of algorithm (nonarchitectural). Some attributes are potentially in conflict with each other, e.g. security and reliability, so that tradeoffs have to be made between them.

**Information sources:** The relevant artefacts for these criteria in the REM model are the quality requirements. It is important to capture them in a way that includes concrete metrics of how to measure the quality and what rating has to be achieved.

**Example:** For embedded systems in general, security is a major concern. Some examples for quality requirements from the navigation system are *robustness* ("How does the system react when it loses the satellite connection?") and *efficiency* is a driving concern as the driver wants to use the system while already on his way and get the resulting route as fast as possible, therefore we have requirements according to available resources, performance, electricity consumption, ...).

## 3.4 Architectural Criteria

The architectural criteria contain communication requirements regarding the interaction with the environment and technical constraints. This type of constraints usually arises bottom-up either from parts of the architecture already defined or from constraints that directly influence the architecture from the technical side (the realization or implementation of the system). These comprise for example certain hardware that is used and cooperating systems that interact with the system to be developed.

**Information sources:** The information regarding the relevant architectural constraints can be found within artefacts from all three content categories of the REM model, according to how early during the process they are assessed and where they arise from. Constraints or conditions from ealier systems or the environment the system has to interact with are included over the different stages of the requirements engineering process and are refined along the way. The General Conditions from the Business Needs will usually be known earlier than the Dependencies from the Requirements Specification. The Design Constraints in the System Specification are either developed bottom-up , or the are refined from the Requirements Specification.

**Example:** For the navigation system, examples for architectural constraints arising from the environment are the given human machine interface (with graphical display and input button), and that the system requires satellite connection.

## 3.5 Decomposing the system according to the criteria

The procedure for decomposing the system according to the introduced criteria can be performed in the following major steps:

1. Identify architecturally significant requirements.
2. Analyse and assign them to the given categories.
3. Consider and apply the criteria according to the order depicted in Fig. 5. There are two general possibilities for the first blueprint:
   - A logical blueprint on the basis of features.
   - A technical blueprint on the basis of architectural constraints.
4. Assure quality of the blueprint.

These steps are explained in more detail to allow for concrete guidance during systems development:

**1.** The preparation for the decomposition is to identify the requirements that qualify as architecturally significant in each category. Jazayeri, Ran and van der Linden give suggestions for detecting them [29, p. 11]:

- "Requirements that cannot be satisfied by one (or a small set of) system components without dependence on the rest of the system. (...)"
- "Requirements that address properties of different categories of components (...)."
- "Requirements that address processes of manipulating multiple components (...). "

**2.** After their identification, the architecturally significant requirements have to be analysed and assigned to the adequate criteria categories directive (Sec. 3.1), functional (Sec. 3.2), quality(Sec. 3.3), and architectural(Sec. 3.4). Within each category, they have to be ordered according to their impact or priority.

**3.** The order of the criterias' consideration and application during the design process is depicted in Fig. 5. There are two alternatives depending on whether there are architectural constraints that in fact predetermine the architecture or not.

(a) If there are, the architecture is already predefined and the functional and quality criteria can only be considered secondarily. This directly imposes or at least restricts the logical design, a coarse-grained blueprint of the system's architecture, leading to solution (1) in the figure.

(b) In the alternative case, when there is no such predetermination, the architecture is organized according to the functionality of the system. This allows us to derive a structured service hierarchy with user-perceptible functions decomposed into realizable subfunctions, depicted by solution (2) in the figure. An analysis identifies parts of functionality that are alike within the different use cases and functional requirements of the system. Those common or alike parts are then abstracted to and grouped and realized as logical components.

We are aware that (a) leads to a less abstract decomposition than (b), because it has a stronger connection to technical constraints, but both solutions primarily lead to a logical blueprint of the architecture.

**4.** After an initial decomposition, either according to architectural or functional criteria, the quality criteria are introduced and evaluated (see also [4]). They can influence the architecture of the resulting system in two ways: Either the structure is modified according to a certain principle, e.g. maintainability. Or some functionality is added to fulfil the requirement, e.g. a component for user identification to satisfy a security requirement.

Although it is possible that a coarse-grained decomposition is taken primarily according to directive critera, it is more usual that especially the organizational criteria influence rather secondarily or in a way that is hard to detect and the companies may even be unaware of it, but this is not depicted in the figure.

**Example:** For the NAVI system, the decision was taken to do a functional decomposition to get a logical architecture independent from the hardware components. This decision is documented in Tab. 2 using a template for architectural design decisions. So we group the functionality in form of interacting logical components as depicted in Fig. 6. The functional requirements are listed in the Requirements Specification in Sec. 2.2.

When analyzing the common parts of functionality within the NAVI use cases (see Fig 2) and functional requirements, we identify the following functional parts:

- "Get GPS data", independent from the data being a given address or the current position of the system,
- "Search for a target", where it does not matter whether the target is a point-of-interest or a concrete address,
- "Manage address book",
- "Manage points-of-interest",
- "Navigate to a destination", again, it does not matter whether the target is a point-of-interest or a concrete address, and
- "Present information to the user", independent from what kind of information it is.

They are abstracted to a certain degree (e.g. from the type of information that shall be presented to the user), grouped and realized as logical components as is depicted in Fig. 6.

The difference to the functional grouping of the requirements in Sec. 2.2 is that they were grouped according to user perception while here, in the logical system specification, they are grouped according to the smaller tasks that have to be performed according to fulfill a certain user-visible function.

## 3.6 Further guidance

In [3] Bass et al. list some tactics to influence quality attributes. The tactics listed are classified into abstract categories. The quality attribute "*Availability*" is subdivided into the categories "*Fault Detection*", "*Recovery-Preparation and Repair*", "*Recovery Reintroduction*" and "*Prevention*". These are general rules for influencing a quality attribute. Within these categories Bass et al. list some elements which adhere to these general rules. The elements are sometimes also rules, heuristics or principles. For the quality attribute "*Modifiability*" the principle "*Hide Information*" is listed under "*Prevention of Ripple Effect*". But most of the times the elements are very concrete architectural means of influencing a quality attribute – patterns. E.g. "*Heartbeat*" is listed as under "*Fault Detection*". If principles and patterns are used to influence the quality attributes of an architecture, then these should somehow be included in the architecture evaluation process. We show how in Sec. 6.

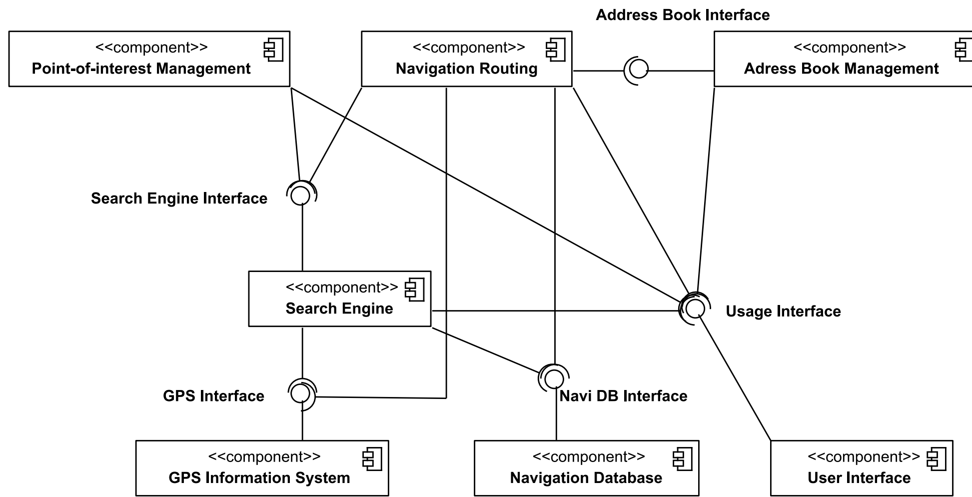


Figure 6: Logical Architecture of the NAVI Example

An extensive collection of patterns can be found for example in [7].

It has to be kept in mind that architecture provides the foundation to the realization of many quality attributes, but it is not possible to achieve them simply through architecture.

# 4 Documentation

Projects that require architecture evaluation work on complex systems [1]. Complex systems often have a long lifetime, their service delivery may be distributed, and they involve many developers who can also be working on different development sites. This implies the need for documentation. Without proper documentation the results are systems that cannot be understood, a design that cannot be reconstructed, and rationale that cannot be retraced. Additionally, the software's quality can not be assessed without immense time expenses. Therefore, for our evaluation method we define appropriate architecture documentation to be a crucial prerequisite.

In Section 4.1 we explain the gaps we see in current techniques of architecture documentation and in Section 4.2 we explain how we define architecture to be documented for our evaluation method.

## 4.1 Shortcomings of Architecture Documentation

In practise the overall architecture of a system is often documented quite minimalistic, for example as UML component diagram or simple box-and-line diagram. As [9] report, "modeling of architectural designs (...) lives on whiteboards, in Microsoft PowerPoint slides, or in the developers' heads". Such a diagram may or may not contain a few descriptive comments, but generally it is scarcely traceable or motivated. So the architecture is represented graphically, but the interpretation is missing.

Industrial partners from current research projects complain that they are frequently missing the rationale that was behind the decisions when a system had been designed. This leads to knowledge leaks and missing information every time an architectural decision is questioned. When the system's quality is assessed, when the system shall be modified or extended, when the architecture has to be reengineered, when a new software developer joins the team and has to make her- or himself familiar with the system — each time the expert who took the decision has to be consulted. This might be only inefficient and time-consuming when the responsible expert is available, but it imposes a serious problem once the expert leaves the company or when it is not clear who took the decision.

Although many companies have realized this gap, no measures have been taken yet to overcome this problem. Neither have new documentation artefacts been introduced nor have existing standard artefacts been extended with respect to this aspect.

In the following section, we define how to document the architecture to make use of our evaluation method.

[1] leaning on [9], we refer to the term "complex system" as a shorthand for "(mainly large-scale) development and support of software-intensive solutions in domains such as defence, aerospace, telecommunication, banking, insurance, healthcare, retail, automotive, etc."

## 4.2 Documenting Layers, Views, and Rationale

To avoid the shortcomings described in Section 4.1, the architecture model we base our reasoning on promotes the ideas of abstraction layers and different views onto those layers. We will explain these ideas in the following, but for further detailed information about the architecture model the reader should refer to [10]. The points we consider as crucial for appropriate architecture documentation within the System Specification (see Sec. 2.3) are abstraction layers, views, rationale, and patterns. Different views on the architecture allow to focus on certain aspects while omitting the others from the representation. However, the model lying behind those specialized representations has to be the same and therefore the views are consistent with respect to each other. The rationale are the causes and reasons that have lead to certain architectural decisions. It is important for tracing, understanding, evaluating, changing, and reusing the architecture or parts of it to have access to that rationale. Patterns are a convenient way to capture best practises and common knowledge in architecture.

**Abstraction Layers:** The architecture is described in three layers of abstraction, each of them in accordance with a certain section of the architecture model that describes the system with a certain degree of abstraction. The degree of abstraction is chosen in such a way that each layer represents especially well certain aspects that are important for the development of complex software systems. The layers are:

- The **(user-)functional unit layer** gives a structured specification of the user-visible functionality. The user can be either a human or another system. The system is seen as black box and the functionality is described by partial services and their relationships.
- The **logical component layer** realizes the services from the above layer via interacting system-internal logical components with equal behaviour. On this layer, the architecture model is enriched by communication channels between the components.
- The **technical implementation layer** shows the software structure, describes the hardware topology, and realizes the software deployment through the realization of application clusters. This layer visualizes differences of the system behaviour due to a certain chosen platform.

Each layer includes a number of views (see next paragraph) which offer appropriate models for those aspects.

**Views:** The IEEE standard 1471 [26] recommends an identification of stakeholders and concerns that shall be satisfied by certain architectural views. Views are a concept orthogonal to the notion of abstraction layers. While perceiving the abstraction layers as zooming in on the system details top-down, the views

are placed horizontally next to each other on the same abstraction layer and allow the explicit focus on a certain aspect of the system design.

The probably best-known work with respect to that topic is Kruchten's 4+1 View Model [37] with the main views logical, process, physical, and development. The ones that are usually present are a structural, a functional, and a behavioural view. [15] promote module views, component-and-connector views, and allocation views. These views can be realized either using UML diagrams (according to [9] the most frequently used notation form), box-and-line diagrams, architecture description languages (ADLs), or executable models with formal semantics.

The views we require to be documented are on the abstraction layer of the logical components:

- Structural view: The white box view on the logical components and how they relate, e.g. depicted by a component model, analogous to Kruchten's logical view, for an example from the NAVI system see Fig.6.

- Interface and border specification: We define an interface description plus a template that captures relevant information with respect to the context or environment of the system, constraints that imply certain restrictions or have led to certain decisions. This view is very important if the logical component is subject to either distributed development or reuse (see also [43]), but it may be neglected if neither applies to the component.

- Behaviour and interaction: The behaviour of the logical components represented e.g. as i/o automata, analogous to Kruchten's process view. The specified behaviour has to be consistent to the behaviour described in the use cases of the Requirements Specification (Sec. 2.2).

- Data definition: All data present in and relevant for the system needs to be documented e.g. using a data dictionary that explains the captured information and its representation, see Tab. 1 for an example from NAVI. This view is closely connected to the interface view, but we emphasize the importance of documenting the semantics and the interpretation of the representation of the data instead of just listing parameters and data types.

- Decision view: The decision templates document the rationale behind decisions that are taken during the development process and can be structured hierarchically, see paragraph below.

Additionally, the documentation has to be consistent among the different representations and shall include the rationale for the concepts selected. The mapping onto the hardware is not within the scope of this report, therefore we do not describe the physical views of the technical implementation layer in detail.

Without prescribing a certain technique, we solely require a representation that makes architectural decisions detectable and quality attributes assessible. For being able to evaluate the architecture, we *always* demand an informal representation of the architecture's structure for communication purposes and *optionally* a formal representation if quantitative tasks like simulation or generation shall be performed. An overview of the architecture has to be given at least in form of a box-and-line diagram that is accompanied by an explaining, meaningful caption. Another possibility are UML models, but we demand the documentation of the exact meaning of the elements if they are used with different semantics than the ones defined in the published UML specification.

| **Data dict** | **Data type** | **Representation** |
|---|---|---|
| Subject | Point-of-interest (POI) | Potentially interesting locations for driver |
| Name | String (max. 60 char) | Individual name of the POI |
| Address | String (max. 120 char) | Street name, city, postcode |
| GPS Data | Integer | Coordinates of the POI |
| Category | Enumeration | Indicates the type of POI (e.g. restaurant) |
| Flag | Bool | Indicates if POI is stored in address book |

Table 1: Data dictionary example for "Point-of-interest" in NAVI

**Rationale:** In order to understand why architectural decisions were taken in a certain way, the relevant directive and architectural criteria must be documented. The choice of representation for an architecture depends on whether the documentation shall serve "only" for communication purposes or also for concerns like simulation or generation that can be performed automatedly. If the latter quantitative tasks are to be performed, it is necessary to use formal methods and notations with strictly defined semantics. In other cases, too formal representations might complicate the understanding. Instead, a rather informal model with an adequate description and sufficient, unambiguous explanation is more appropriate.

The decision view is the hierarchical structure of all taken decisions, each of them documented in a standardized way with links to other related decisions. [20] suggest an implementation as hyperlinked documentation on top of other views.

Facing the issue of rationale documentation, [47] suggest to "demystify" architecture by using decision description templates. The decision description template depicted in Tab. 2 is inspired by [47] and captures the rationale around a design decision. It records what the decision is about, the background, who participated in taking it, alternatives and their arguments as well as related decisions, requirements and artefacts that the decision impacts. The latter include the architecture specification with the above mentioned views, which

| **Architecture decision** | |
|---|---|
| Issue | Navigation system coarse-grained decomposition |
| Decision | Functional decomposition is dominant |
| Status | Approved by board |
| Group/Stakeholder | Paul Miller (software architect), Mike Shaw (product manager) |
| Assumptions | We want to reuse parts of the functionality in later systems. We want to be independent of hardware details. We need to develop in parallel to save time. |
| Constraints | None |
| Positions | Decomposition without hardware details; hardware-close decomposition |
| Argument | If we decompose the system into logical functional units, we get a logical architecture independent from the hardware, so it won't be affected by changes in the hardware and can be reused in parts or set up the base for a product line. |
| Implications | The functional units can be developed in parallel. We can provide the system for different platforms. The component "GPS Info System" is independent of hardware details and data formats. |
| Related decisions | This decision is the root architectural decision, all others relate. |
| Related Requirements | Business Needs (Sec. 2.1): Different platforms, independence of supplier, increase reuse to save money. |
| Related Artefacts | Logical architecture structural view (Fig. 6). |
| Related principles | Planned reuse improves the quality and saves money. |
| Notes | None. |

Table 2: Architectural decision description template

we document by visualizing it in form of model diagrams and by describing used patterns.

The crucial point for architecture evaluation is to provide a representation that *allows to recognize architectural decisions* and that provides the possibility to assess these decisions' effect on quality attributes.

# 5 Shortcomings of Common Architecture Analysis Methods

Architecture analysis methods can be classified into *quantitative* and *qualitative* analysis methods [44][2]. While the quantitative methods produce values which can be compared on an ordinal or even interval or ratio scale, the qualitative methods use expertise to discover risks or misconceptions.

Although the majority of quantitative architecture analysis methods are based on formal foundations (e.g. composition rules) and systematic, repeatable and traceable, it is only possible to determine the expected value of a single quality attribute. E.g. FTA [48]. Moreover, alternatives are not considered. If the result of an analysis has a satisfying value (e.g. the availability of the system is above 95%), the overall availability quality requirement might be satisfied, but an alternative architecture in which the availability might also be satisfied combined with a better maintainability would never be considered. Furthermore, some quality attributes, such as maintainability, are hard to express in numbered values. And the composition of these values is disputable (the overall maintainability of a system with two components is hard to determine by combination of the maintainability vaules of each of the components).

Qualitative architecture analysis methods address these shortcomings. However, qualitative methods are heavily based on experts performing the evaluation based on their own experience. At present, the most well known qualitative method to evaluate architectures is ATAM [34, 16]. In the following we will analyse the systematic guidance that ATAM provides and show that the parts of ATAM which guide the architecture evaluation (and are not centered on requirements elicitation or social engineering) are mostly based on the guidance provided by experts conducting the ATAM evaluation (Section 5.1). We conclude by deriving goals for our new evaluation method in Section 5.2.

## 5.1 Analysis of the Systematic Guidance for Architecture Evaluation in ATAM

ATAM is organized into four phases. Phase 0 “is a setup phase in which the evaluation team is created and a partnership is formed between the evaluation organization and the organization whose architecture is to be evaluated” [16, p. 70]. The last phase “is a follow-up phase in which a final report is produced, follow-on actions (if any) are planned, and the evaluation organization updates its archives and experience base” [16, p. 70]. Both phases provide systematical guidance concerning organization and social engineering. The remaining two phases present the core of ATAM which is split up into nine steps. We will give brief explanations for each step in the following. For more detailed descriptions the reader should refer to [16].

1. *Present the ATAM.* The ATAM is explained to the participants.

[2] Abowd et al. use the classification terms *measuring* and *questioning techniques* correspondingly [1].

2. *Present the business drivers.* The reason for the development effort is explained to the participants.

3. *Present the architecture.* The architect explains the architecture and how it addresses the business drivers.

4. *Identify the architectural approaches.* The evaluation team captures any architectural approaches based on information by the architect and on the expertise of the members of the evaluation team.

5. *Generate the quality attribute utility tree.* The evaluation team helps the stakeholders to elicit, prioritize and structure the quality requirements with the use of scenarios (as defined in [3]).

6. *Analyse the architectural approaches.* The scenarios of step five are correlated to the architectural approaches identified in step four. If the use of an architectural approach has or may have an influence on the response measure of a scenario[3], it will be marked as a sensitivity or (if it influences more than one scenario) tradeoff point. Depending on the expected influence on the response measures of required scenarios the sensitivity and tradeoff points are categorized as risks or non risks.

7. *Brainstorm and prioritize scenarios.* This step is a reiteration of step five involving a lager group of stakeholders for eliciting and prioritizing scenarios.

8. *Analyse the architectural approaches.* This step is a reiteration of step six with the new scenarios of step seven.

9. *Present results.* The results of the conducted analysis are presented and delivered.

Steps 1-3 and 9 are presentations. ATAM provides guidance as on how and what to present in these steps. Again, the guidance provided is in the area of organization and social engineering. In step 4 the architectural approaches are identified relying exclusively on the expertise of the evaluation team and architect. Step 5 is used to elicit quality requirements and provides methodical guidance in social and requirements engineering. Finally, steps 7 and 8 are reiterations of steps 5 and 6. Thus, the part of ATAM where the architectural evaluation takes place is contained solely in step 6, where the correlation between requirements and architectural specification takes place. The methodical guidance provided by ATAM in this step is confined to the categorization into sensitivity and tradeoff points, risks and non-risks, and some guidance on how to identify these through analysis questions. Hence, even though methodical guidance is given, this step is still heavily based on the evaluation team's expertise.

[3] refer to [3] for further details on the meaning of response measures for scenarios

The guidance provided by ATAM in the areas of organization, social and requirements engineering is crucial and probably a success factor of ATAM. However, it is not specific to architecture evaluation and could probably be successfully applied to other areas of software engineering in which stakeholders have to interact and agree on decisions (e.g. requirements engineering). The guidance provided by ATAM in architecture evaluation is helpful, but not sufficiently elaborated and relies on the participation of experts. The same applies to other well-known qualitative architecture evaluation methods [21].

## 5.2 Goals for a New Qualitative Architecture Evaluation Method

From the analysis of the shortcomings of common architecture evaluation methods we have derived a set of goals for the new qualitative architecture evaluation method which we developed. The goals were,

- G1 to reduce the need for experts within the architecture evaluation method,
- G2 to enhance the systematic process, repeatability and traceability,
- G3 to enhance the integration of the architecture evaluation into the development process,
- G4 to define interfaces to quantitative architecture evaluation methods,
- G5 to enhance the systematic reuse of the knowledge obtained when performing an architectural evaluation in order to improve further architectural evaluations and
- G6 to move the focus of architectural evaluation away from requirements elicitation, social engineering and retroactive architecture documentation towards the actual purpose of architecture evaluation – the determination of how architecturally significant requirements have been addressed by the architecture and its architecturally significant decisions.

The numbering and ordering of the goals are for reference only and have no meaning relating to prioritization.

# 6 Evaluating an Architecture Using POSAAM

According to the SARA Report "The purpose of an architecture review is to understand the impact of every architecturally significant decision (ASD) on every architecturally significant requirement (ASR)" [42]. Hence, to perform any sort of evaluation it is possible to go through every ASD and try to find out how it impacts every ASR or to go the other way around and go through every ASR and find out how the ASDs affect these.

In our evaluation approach we decided to go from ASRs to ASDs for two main reasons: Firstly, going from ASDs to ASRs automatically leads to checking whether the architecture satisfies the requirements. This is exactly what quantitative methods do. Yet, for a qualitative architectural evaluation this is not enough. It is necessary to find out whether all architectural alternatives have been considered and whether the choices made lead to not just a satisfaction of single requirements, but an optimization of the combination of all ASRs. Secondly, going through the ASDs discovered in architectural descriptions implies having an expert evaluator who is familiar with the architectural approaches used, since it would not be possible for a non-expert to identify architectural approaches unknown to him. Starting off with ASRs makes it easier to perform a systematic search for ASDs and thus reduces the need for an expert evaluator.

The main technique used by POSAAM to reduce the need for an expert evaluator is to make use of the expertise encapsulated in patterns within the evaluation process. [14, p. 8] define a pattern as follows:

> *A* pattern for software architecture *describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.*

Presently the knowledge encapsulated in patterns can only be used by those who are already familiar with the patterns. By associating patterns with requirements the search for patterns can be narrowed, as we show in Section 6.1. Patterns are organized into sets of interacting patterns such as for example pattern languages. These can help in searching for further patterns as illustrated in Section 6.2. After a successful search for a pattern, the knowledge in these patterns can be used to perform checks of the architectural decisions taken. The concepts we propose are explained in Section 6.3. Finally, in Section 6.4 some special cases of the evaluation mechanism are laid down, such as how to proceed if a pattern cannot be found.

Opposed to other architecture evaluation methods, we do not explain POSAAM in steps. The reason for this is that POSAAM is not linear. It contains several case differentiations. These in turn are very fine grained and were conceived as procedural support to evaluators. Therefore, in the following we provide the theoretical foundations that led our design of POSAAM.

## 6.1 Moving from Requirements to Patterns

POSAAM is based on the assumption that in software based systems quality requirements are mostly of the same kind (as within ISO 9126) and are also implemented using the same concepts in the majority of cases. Or – as Schumacher et al. express it for general problems: “It is not often that a new development project tackles genuinely new problems that demand truly novel solutions” [46, p. 2].

A quality requirement can be classified into quality attributes and quality sub-attributes. Further refinements are possible. Those patterns, which are designed to influence quality attributes, can be classified according to the quality-attributes they are intended to influence. Through this classification the quality requirements and the patterns which can be used to influence these quality requirements can be linked. [46] is a collection of patterns which all belong to the classification “Security”. The pattern descriptions implicitly contain information which can be used for further classifications. Using this information an evaluator can go through the ASRs one by one and obtain a set of patterns which could potentially be used to address the ASR being observed. An example using patterns from [46] follows:

The quality requirement Q1 explained in our example from Sec. 2 on page 9 can be classified within a quality attribute tree (e.g. according to ISO9126 [30]) as “*Quality requirement; Security; Privacy*”. This node in the tree is associated with the patterns “*Authorization*”, “*Role Based Access Control*” and “*Multilevel Security*” from [46]. If the architecture, which is to be evaluated, implements one of these patterns, we assume that an evaluator can identify it. The subsection on “Special Issues” (6.4) shows how to proceed if the architecture does not implement any of the patterns or if the evaluator cannot identify any of the patterns.

## 6.2 Networking: Using Pattern Relations

Patterns exist in relation to other patterns [11, 13]. For our purposes we will slightly modify the naming and definition of relations mentioned in [11, 13]. We consider a specialization relation “*is a*” (1), meaning that a pattern can be a specialized form of a more general pattern, a dependency relation “*needs*” (2), meaning that the use of a pattern implies the use of another, and a generic relation “*uses*” (3), meaning that the use of a pattern is often combined with further patterns. The refinement relation means that the implementation of a pattern is tackled using further patterns. At present the refinement relation is only considered as a special case of the generic relation. Other forms of pattern relations will not be considered within the scope of this paper.

The dependency relation (2) implies the use of another pattern. Hence, if an evaluator discovers a pattern which has a dependency relation to another pattern then the second pattern must also be part of the architecture. In our example from above this would mean that once an “*Authorization*” pattern has been identified, an “*Identification and Authentication*” pattern is also necessary.

The generic relation (3) has similar implications. The evaluator can use the information about the generic relations to check whether related patterns have been, could or should have been used. Naturally, the information when which pattern is to be used must be available to the evaluator. This information is often found implicitly in pattern languages.

If a pattern that has been identified in an architecture, which is to be evaluated, has one or more specializations (1), it is necessary to check whether the decision for using the pattern itself or one of its specializations has been taken explicitly, whether it has been documented and whether the decision was taken correctly. Again, in order to be able to perform this check, information from pattern languages can be used. In our example the patterns "*User ID/Password*", "*Biometrics*", "*PKI*", "*Hardware Token*", and "*Unregistered Users*" are all specializations of a generic "*Identification and Authentification*" pattern. In [46, Section 7.2 "*Automated I&A Design Alternatives*"] the trade offs of using these patterns are explained. The provided information on the trade offs can be used by an evaluator to determine whether the appropriate pattern has been chosen.

The presented relations between patterns define a strict procedure for the evaluation process which is independent of the evaluator performing the evaluation. By making use of a knowledge base containing information about pattern relations the procedure will lead to the same results (the same patterns to be observed in the same order) within the evaluation process.

## 6.3 Using Pattern Knowledge for Evaluation

Once a pattern has been identified the first check which is to be performed is its correct implementation, including the necessary components assuming the required responsibilities and correct communication structures. Depending on the form of the documented architecture these checks may imply the search for a partial graph in the graph of components. However, the responsibilities of components are usually not modeled in the architectural decomposition. Furthermore, the components of the pattern are not always in the same abstraction layer of the hierarchical decomposition of the modeled architecture. Hence, the search cannot be automatized and needs to be performed by a human. Nevertheless the person performing the search should be able to identify the patterns necessary components using the architectural documentation without further support by an architect who designed the system, because otherwise the documentation is of no use. After the check for the correct implementation of the pattern the actual evaluation takes place.

Patterns encapsulate a lot of expertise, including an implicit description of the sensitivity and trade off points contained within the pattern (usually in the forces section) and information as on when which configuration of these makes sense. The evaluator can take advantage of this information to check whether the right decisions have been taken.

Even for such a seemingly simple pattern such as "User ID/Password" there are a lot of decisions to take. Which characters are acceptable? What length range is acceptable? Who can create new passwords? What is the lifetime of a

password? And several more (see [46, pages 219 and following]).

The knowledge contained in pattern descriptions can also be used to identify risks. Usually this is achieved through the identification of decisions that were not taken or documented explicitly, but also if the decisions taken do not comply with stated requirements. Furthermore the pattern descriptions aid in refining requirements, as refinement of design will always go hand in hand with refinement of requirements. Often the decision that has to be taken was not and could not be taken, since the information necessary to take the decision was not available. In this case the evaluation leads to the refinement of requirements and design simultaneously.

We summarize the steps explained up to now in an activity diagram in Figure 7. The activity "*perform pattern specific checks*" includes searching for further patterns using pattern relations.

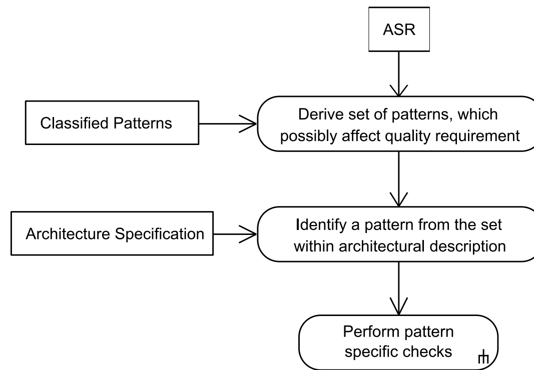


Figure 7: The three main steps in POSAAM

## 6.4 Special Issues

POSAAM is based on the identification and use of patterns as a means of evaluation. However, we are aware of the facts that (1) patterns are not always available for every problem, (2) patterns may not always present the best solution to a problem, (3) patterns may not always be easy to identify in architectural descriptions and (4) POSAAM can only be applied if the patterns are structured and stored in a form that supports the evaluation. Figure 8 is an extension of Figure 7. The numbered objects of the diagram correspond to the issues listed above. Explanations on how to handle the issues follow.

### 6.4.1 Evaluating Using Principles

POSAAM addresses the first issue (1) by relying on principles. As we have illustrated in Section 3, all forms of influencing quality attributes through architectural design rely on some sort of principles, rules or heuristics. In POSAAM we subsume all of these under the term principles. Just like patterns, principles

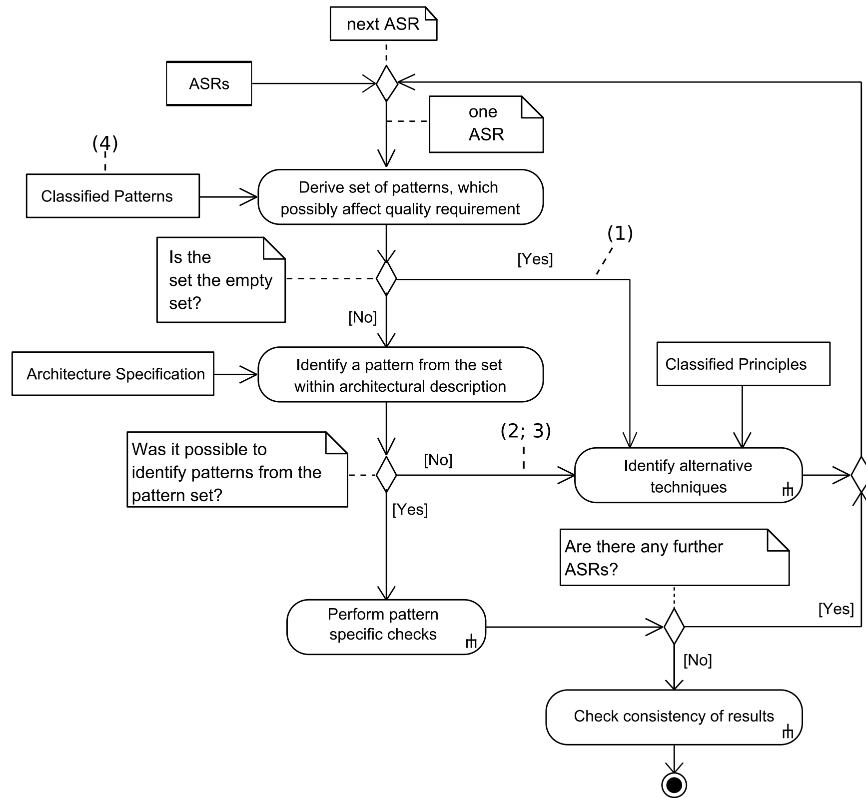


Figure 8: The core of the POSAAM evaluation process

used to influence quality attributes can be classified within a quality attribute tree. Hence, if a pattern is not known / not available for an ASR, the evaluator can check for the fulfilment of the corresponding principle. In this case some more expertise is required from the evaluator. Still the evaluation is lead by the conformance to principles, which makes it more systematic and traceable than to simply "connect" requirements to design decisions.

That a pattern cannot be identified within the architectural description provided for evaluation can have two different reasons: The architect may have chosen not to use the pattern (2) or the pattern is not visible to the evaluator within the architectural documentation available (3). To POSAAM this does not make a difference as in both cases the evaluator uses the knowledge provided by principles and the patterns applicable to the requirement being processed to proceed with the evaluation. After identifying a pattern applicable to the requirement being processed, the evaluator can use the decisions that would have to be taken when applying the pattern to perform the evaluation. If the decisions that would have to be taken when applying the pattern are documented in the architecture specification, it is irrelevant whether the architect actually

applied the pattern or not, as long as the necessary decisions have been made. However, if the decisions have not been addressed and the evaluator cannot understand why, i.e. the evaluator cannot trace other decisions to the principles that can be used to influence the requirement, then either the architectural documentation or the architecture itself have a quality deficit. Both cases present an appropriate result for an architectural evaluation.

If the evaluator identifies an alternative technique to address an ASR, the alternative should be analysed to find out whether the technique might be generalized for other problems. The solution used in the architecture might well be a new pattern. But since patters represent proven solutions to a problem [13] it would not be wise to add the identified solution as a pattern to the collection of patterns. Therefore, POSAAM introduces the concept of a pattern candidate, which can be used as an intermediate step towards adding a pattern to the pattern collection.

### 6.4.2 Structuring Expertise Stored in Patterns

For the application of POSAAM the information implicitly stored in patterns needs to be made explicit (4). The pattern community does not fail to point out that patterns are written for humans and not machines. Nevertheless, we believe that a systematic process, even though conducted by humans, has a strong resemblance to an algorithm. And as such its data representations should be laid out accordingly, since data representations are "*...where the heart of a program lies*" [8]. This especially applies when the processing unit (in the case of our process, a human being) is slow and tends to non-deterministic behaviour.

To this purpose *Malich* [40] proposes a table storing the correlations between patterns and quality attributes (see Fig. 9). The columns of the table are sorted according to quality attributes and the rows of the table according to patterns. The cells contain explicit information on the influences of the pattern on the quality attribute of the corresponding column. Such information may be constructs of the pattern which have a positive or negative influence on the respective quality attribute, but also constructs of the pattern which present sensitivity or tradeoff points of the respective quality attribute.

A similar approach is used by *Klein et al.* in their publications about Attribute Based Architectural Styles (ABAS) [35, 36]. *Klein et al.* combine reasoning about quality attributes with patterns. Every ABAS provides a reasoning framework on how an architectural style (pattern) influences a quality attribute. For some quality attributes the reasoning framework may even be a formal model, such as a Markov model for the availability of a Leader-Backup pattern (for further details on this example we refer to [36]).

POSAAM builds on both of the presented approaches. However, for the use of POSAAM further information about patterns needs to be obtained and stored in a structured form. The first step in restructuring the information contained in patterns is to provide a new definition of the term "pattern", which is slightly different from the prevalent definition given by [14] which we quote at the beginning of this section.

Figure 9: A table correlating patterns to quality attributes ([40])

> *A pattern for software architecture defines a configurable well-proven solution to a problem class of recurring design problems. The pattern consists of recurring as well as variable parts. The pattern describes which parts are recurring, which parts vary, which variations are possible, and which effects can be accomplished by the variations. The description of the parts of a pattern (either recurring or varying) comprise components, their responsibilities and relationships and the ways in which they collaborate.*
>
> *A solution to a design problem belonging to the problem class of pattern* P*, which includes the recurring parts of pattern* P *and a variation of the varying parts as defined by pattern* P *is called a configuration of pattern* P*.*

Through the use of this definition the parts belonging to a pattern can be described in greater precision. With the information about the recurring and varying parts and the information about configurations of patterns the pattern specific checks explained in Section 6.3 can be performed in more detail. Apart from checks about the correct components, responsibilities, relationships, and collaborations of recurring and varying parts, checks about the configurations include whether the configuration is appropriate to the given quality requirements. Using this definition both the sensitivity and trade off points are always dependent on a variation point. Storing sensitivity and tradeoff points as dependent on a variation point enables evaluating an architecture by associating the effects of each variation point of a pattern with the quality attributes of the resulting system and the quality requirements.

The definition also allows to store information about possible risks. A risk in a software architecture is a combination of architecturally significant decisions which may impede the achievement of an architecturally significant requirement. It is possible to add information about configurations of variation points which lead to a restriction of quality attributes. These may be characterized as risks.

The concept of ABAS can also be combined with this definition. The reasoning framework given in an ABAS only needs to be correlated with the variation points of the pattern.

The pattern itself may be correlated to one or more principles (as Bass et al. point out in [3]). Correlating patterns to principles helps in the search for alternative implementations to architecturally significant decisions, even when a pattern cannot be identified in the architectural specification given.

Finally, the relationships between the patterns can be stored separately in a graph. The graph contains information about the possible alternatives and when which alternative is more suitable.

As POSAAM makes use of principles for evaluation, principles also need to be stored and structured. The information necessary to be able to perform a POSAAM evaluation is a correlation of principles to quality attributes. This can be achieved through a table just as the one used for the storage of patterns.

Further information that should be stored is information about when to use quantitative architecture evaluation methods and information that supports the following steps of development. Such information may be constraints for the implementation or even evaluation specifications which may be applied after implementation of the pattern.

# 7 Related Work

In this section, we give a short overview over the related work in the three categories belonging to the respective chapters of this work: design, documentation, and evaluation. An overview over the related work of patterns and extraction of information from patterns is given in a fourth category.

**Design:** In [3, Chapter 5] Bass et al. dedicate a whole chapter to influencing quality characteristics of software systems through architectural tactics. Attribute-Driven Design [49] focuses mainly on satisfying quality requirements that were identified as key performance indicators beforehand in workshops with different stakeholders. We delineate our work by explicitly taking into consideration all potential influences on the decomposition of a system and describing their application.

**Documentation:** Nava et al. [41] present a prototype tool for creating and exploiting architectural design decisions. [24] propose to combine the documentation of decisions and patterns. [26] describe the documentation rather abstract but based on a common conceptual model of architectural description with views and viewpoints. [15] define view types that can be instantiated to views with the adequate degree of detail as documentation artefacts. As opposed to those works, our approach encompasses an artefact model based on a combination of abstraction layers and views with interconnection via decision rationale.

**Evaluation:** By now there are a whole range of qualitative architecture evaluation methods. Besides the most well-known ATAM [33, 34, 16], SAAM [32, 31, 16], and ALMA [5, 38, 6] there are further qualitative evaluation methods which have been compared in the taxonomies by Babar et al. [2], Dobrica and Niemelä [19], and Eicker et al. [21]. The SARA Report [42] is a summary of best practices in the evaluation of software architectures. The proposed method POSAAM differs in explicitly not relying on expert knowledge during the review but instead identifying and analysing patterns and principles.

**Patterns:** There are numerous collections of patterns and pattern languages, the most important being the Design Patterns book by the "Gang of four" (Gof) [22] and the Pattern Oriented Architecture (POSA) series [14, 45, 28, 12, 13]. Extracting the and reorganizing the information stored in patterns for evaluation purposes has been reasoned about in the works of Zhu et al. [50], Klein et al. [36, 35] and Malich [40, 39]. Zhu et al. extract and reorganize general scenarios and architectural tactics from patterns. With their concept of attribute based architectural styles (ABAS) Klein et al. provide a reasoning framework for the quality attributes influenced by patterns. Malich correlates patterns to quality attributes and uses the concept of sensitivity and tradeoff points to reason about

pattern quality. In [51] *Zimmer* analyzes relationships of the design patterns from the Gof Book.

# 8 Conclusion

Within this report we present a new supporting and extending architecture evaluation method, which we derived from the shortcomings of presently known architecture evaluation methods. In order to determine how to evaluate architectures we first explain our understanding of how to construct and document architectures.

The artefact model we use to document the architecture unifies requirements engineering and design. We are currently working on an extensive manual for integrated requirements engineering and design for software-intensive embedded systems in cooperation with industrial partners (automotive original equipment manufacturers and first tier suppliers). The corresponding process described in the manual is artefact-centric and based on an artefact model close to the one presented in this paper.

The motivation for creating a new qualitative architecture analysis method lies in the gap contained in known architecture analysis methods, which is elegantly closed by relying on experts during the evaluation process. Needless to say that experts will always be necessary to perform qualitative evaluations, nevertheless the need for an expert can be diminished by capturing, structuring and referring to expert knowledge. POSAAM is a new method to evaluate the architectures of software intensive systems using the expertise encapsulated in patterns. Thus, making the evaluation more systematic, repeatable and traceable. An architect preparing for a POSAAM evaluation knows what to expect and can prepare the architecture documentation correspondingly and will have a better understanding of evaluation results.

POSAAM complements the existing architecture evaluation methods in that it presents a refinement at the stage where ASRs have to be put into relation with ASDs.

Before presenting our new architecture evaluation method, we provided an analysis of presently known architecture evaluation methods which resulted in a set of goals for the new evaluation method (see Sec. 5.2 on page 26). In the following section 8.1 we describe how these goals are addressed by POSAAM. The analysis of how the goals have been addressed by POSAAM reveals areas for further work which will be mentioned separately in Section 8.2.

## 8.1 How the goals of POSAAM have been addressed

The main technique used by POSAAM to reduce the need for an expert evaluator (G1) is to make use of the expertise encapsulated in patterns within the evaluation process. However, without a procedure that helps in identifying patterns relevant for evaluation, the knowledge in patterns can only be used by evaluators who are already acquainted with these patterns. The systematic guidance in the search for patterns explained in the introduction of Section 6

and in Section 6.1 therefore addresses goals G1 and G2 (enhance systematic process).

The relations between patterns and the corresponding procedure for the evaluation process also decrease the level of dependence of an expert evaluator (G1). By making use of a knowledge base containing information about pattern relations the procedure will lead to the same results (the same patterns to be observed in the same order) when performed by different evaluators. Thus, making the evaluation process repeatable and traceable as declared in the goals G1 and G2.

Goal G6 (shift focus away from requirements elicitation, social engineering and retroactive architecture documentation) is mainly addressed by starting the evaluation process with ASRs and an architectural specification which are required to be available in a specified representation before the POSAAM evaluation can begin. Naturally it is also important to correctly obtain these, but the focus of the POSAAM process description is laid in the determination of how architecturally significant requirements have been addressed by the architecture specification. Since the architecture specification and requirements are required to be available for a POSAAM evaluation, the evaluation becomes a technical process, reducing the need for the social engineering parts of the evaluation presented in other architecture evaluation methods.

The concept of pattern candidates introduced in Section 6.4 addresses goal G5 (reuse of evaluation knowledge). However, a detailed description of the reuse mechanism is still missing and will be addressed by further research.

Accordingly, goals G3 (integration of evaluation into development process) and G4 (interfaces to quantitative architecture evaluation methods) have not been described within this document. The idea for G3 is to integrate information about further quality assurance methods (such as how and when a component should be tested) into the pattern descriptions. Analogous, information concerning the use of alternative architecture evaluation methods could also be included in pattern descriptions. Both goals have not been elaborated sufficiently and need to be reseached further.

## 8.2 Further work

The POSAAM process has already been detailed into more fine grained steps and case differentiations, but further refinements are being worked on. These refinements especially concern the systematic maintenance and enhancement of an evaluation knowledge base (G5).

POSAAM needs to be validated on a real life project. First efforts are being taken in this direction.

The reorganization of pattern descriptions which we propose for the use in POSAAM also needs to be refined and cast into a semi-formal definition. Especially patterns need to be classified according to the quality attributes they are intended to affect, integration of further quality assurance methods (G3) and of integration points for quantitative evaluation methods (G4).

The knowledge base and the evaluation process can be supported through the use of software tools. The concepts for this purpose already exist, but a prototypical implementation would provide further insights.

# References


[1] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrop, and A. Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, CMU, SEI, 1997.

[2] M. A. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In L. Zhu, editor, *Australian Software Engineering Conference, 2004. (ASWEC'04)*, pages 309–318, 2004.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, second edition edition, 2003.

[4] P. Bengtsson and J. Bosch. Scenario-based Software Architecture Reengineering. In *International Conference on Software Reuse. Proceedings.*, pages 308–317, 1998.

[5] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Analyzing software architectures for modifiability. Technical Report HK/R-RES—00/11—SE, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Sweden, S-372 25 Ronneby, Sweden, 2000.

[6] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004.

[7] G. Booch. Handbook of software architecture. Website, 2008. http://www.booch.com/architecture/index.jsp.

[8] F. P. Brooks. *The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition.* Addison-Wesley Publishing Co, 1995.

[9] A. W. Brown and J. A. McDermid. The art and science of software architecture. In F. Oquendo, editor, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2007.

[10] M. Broy, M. Feilkas, D. Wild, J. Hartmann, J. Grünbauer, A. Gruler, and A. Harhurin. Umfassendes Architekturmodell für das Engineering eingebetteter software-intensiver Systeme. Technical report, Technical University of Munich, will be published in 2008.

[11] F. Buschmann, K. Henney, and D. C. Schmidt. Past, present, and future trends in software patterns. *IEEE Software*, 24(4):31–37, July-Aug. 2007.

[12] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, volume 4. John Wiley & Sons, Inc. New York, NY, USA, 2007.

[13] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley & Sons, Ltd, 2007.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*, volume 1. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[15] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, and R. Little. *Documenting Software Architecture*. ISBN:0201703726. Addison-Wesley, 2003.

[16] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.

[17] A. Cockburn. The interaction of social issues and software architecture. *Commun. ACM*, 39(10):40–46, 1996.

[18] M. Conway. How do committees invent? *Datamation Journal*, pages 28–31, April 1968.

[19] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *Transactions on Software Engineering*, 28(7):638–653, 2002.

[20] J. C. Dueñas and R. Capilla. The decision view of software architecture. In R. Morrison and F. Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 222–230. Springer, 2005.

[21] S. Eicker, C. Hegmanns, and S. Malich. Auswahl von Bewertungsmethoden für Softwarearchitekturen. In *ICB-Research Report*, number 14 in ICB-Research Reports. Universität Duisburg Essen, March 2007. (in German).

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] E. Geisberger, M. Broy, B. Berenbach, J. Kazmeier, D. Paulish, and A. Rudorfer. Requirements Engineering Reference Model (REM). Technical report, Technische Universität München, 2006.

[24] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Software*, 4:38–45, 2007.

[25] J. D. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 85–95, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[26] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (ANSI/IEEE-Std-1471)*, ieee-sa standards board edition, September 2000.

[27] International Standardization Organisation. *ISO9126 - International Standard for the Evaluation of Software Quality*, 2001.

[28] P. Jain and M. Kircher. *Pattern-Oriented Software Architecture: Patterns for Resource Management*, volume 3. John Wiley & Sons, 2004.

[29] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families*. Addison-Wesley, 2000.

[30] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 7, Software engineering. *ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model*. International Organization for Standardization and International Electrotechnical Commission, June 2001.

[31] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55, 1996.

[32] R. Kazman, L. Bass, M. Webb, and G. Abowd. SAAM: a method for analyzing the properties of software architectures. In *Proceedings of the 16th international conference on Software engineering*, pages 81–90. IEEE Computer Society Press Los Alamitos, CA, USA, 1994.

[33] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *Fourth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS '98. Proceedings.*, pages 68–78, 1998.

[34] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, CMU, SEI, 2000.

[35] M. Klein and R. Kazman. Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-022, CMU, SEI, 1999.

[36] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architecture Styles. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 225*, volume 243, 1999.

[37] P. Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[38] N. Lassing, P. Bengtsson, H. van Vliet, and J. Bosch. Experiences with ALMA: Architecture-Level Modifiability Analysis. *Journal of Systems and Software*, 61(1):47–57, Mar. 2002.

[39] S. Malich. *Ein pattern-basiertes Wissensmodell zur Unterstützung des Entwurfs und der Bewertung von Softwarearchitekturen.* PhD thesis, Universität Duisburg-Essen, Oct. 2007.

[40] S. Malich. *Qualität von Softwaresystemen: Ein pattern-basiertes Wissensmodell zur Unterstützung des Entwurfs und der Bewertung von Softwarearchitekturen.* Gabler Edition Wissenschaft, May 2008.

[41] F. Nava, R. Capilla, and J. C. Dueñas. Processes for creating and exploiting architectural design decisions with tool support. In F. Oquendo, editor, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 321–324. Springer, 2007.

[42] H. Obbink, P. Kruchten, W. Kozaczynski, R. Hilliard, A. Ran, H. Postema, L. Dominick, R. Kazman, W. Tracz, and E. Kahane. Report on Software Architecture Review and Assessment (SARA), Feb. 2002. Version 1.0.

[43] B. Penzenstadler and D. Koss. High confidence subsystem modelling for reuse. In *Intl. Conf. for Software Reuse*, 2008.

[44] R. Reussner and W. Hasselbring, editors. *Handbuch der Software-Architektur.* dpunkt.verlag, 2006. (in German).

[45] D. C. Schmidt, H. Rohnert, M. Stal, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Inc. New York, NY, USA, 2000.

[46] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering.* John Wiley & Sons, Ltd, 2005.

[47] J. Tyree and A. Akerman. Architecture decisions: demystifying architecture. *Software, IEEE*, 22(2):19–27, March-April 2005.

[48] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook. Technical Report NUREG-0492, United States Nuclear Regulatory Commission, Washington, D.C. 20555, Jan. 1981.

[49] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-driven design (add). Technical Report CMU/SEI-2006-TR-023, CMU, SEI, 2006.

[50] L. Zhu, M. A. Babar, and R. Jeffery. Mining patterns to support software architecture evaluation. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 25–34, June 2004.

[51] W. Zimmer. Relationships between design patterns. In *Pattern languages of program design*, pages 345–364. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.