

Designing High Quality Architectures

Mari Matinlassi and Eila Niemelä
VTT Electronics, Embedded software

Software Architectures Group
P.O. Box 1100
FIN-90571 Oulu, Finland
+358 08 551 2111

{Mari.Matinlassi, Eila.Niemela}@vtt.fi

ABSTRACT

This paper presents a quality-driven software architecture design method. The method produces architectural descriptions at two levels of abstraction. Both the levels are described using four similarly named views: structural, behavior, deployment and development. More importantly, the method enables the realization of quality attributes by means of quality requirements and quality scope, which leads to the selection and localization of architectural styles and patterns. Quality attributes can be traced from the quality requirement definitions to architectural descriptions, wherein each of the views concerns certain quality attributes.

Keywords

Software quality, architecture, design methods, QAD

1. INTRODUCTION

Software *quality* is the totality of characteristics of an entity that contribute to its ability to satisfy stated and implied needs [12]. Software *architecture* is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment [11]. Software architecture also includes the principles guiding its design and evolution, and therefore, it has a strong influence over the life cycle of a system. The allure of a careful software architecture design is that appropriately selected software architecture increases the quality of a software product.

In the past, hardware engulfed other aspects of a system, and quality attributes in particular, like modifiability, interoperability and reusability were sacrificed first in the course of system development. Today, software-intensive systems are pervasive. The increasing complexity and size of software, as well as the cost of software development and more mature software technologies have changed the role of software architecture.

Architectures do not appear incidentally. Instead, defined processes and methods are required. Earlier applied methods [10],

[13], [14], [15] are capable and exhaustive in their own way; but none of them highlights the quality-driven aspects of architectural design. The Architecture Based Design (ABD) method [1] is a quality driven method for designing software architecture for a long-lived system at the conceptual level of abstraction. In ABD, the conceptual architecture is a representation of the high-level design choices described with three architectural views. In spite of the ABD method having been developed further into a new method called the Attribute Driven Design method, ADD [3], it still does not provide more than a coarse grained high-level, i.e. conceptual architecture as an output.

This paper represents a quality-driven architecture design (QAD) method that provides a systematic way to transform functional and quality requirements for software architecture. The method also utilizes styles and patterns as a guide to carry out quality requirements in architectural descriptions with documented design rationale. Therefore, quality can be traced through the design; from the requirements to high-quality architectures.

2. QUALITY-DRIVEN ARCHITECTURE DESIGN METHOD

The Quality-driven architecture design method (QAD) [16] provides a means of turning the system quality and functional requirements into software architecture. Architectural styles and patterns serve as essential elements in reaching the requirements. The quality of the architecture is assessed by the architectural quality analysis [7], [8] and thereafter, required changes are updated to architectural models (Figure 1).

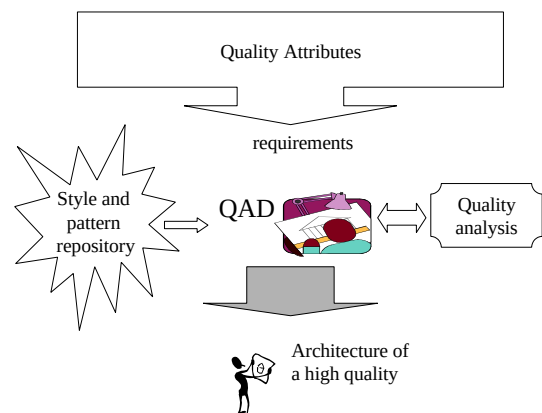


Figure 1. Designing high-quality architectures.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE'02, May 19-25, 2002, Orlando, Florida.

Copyright 2002 ACM 1-58113-000-0/00/0000...\$5.00.

When designing software architectures it is not feasible to begin with the bottom-up style because the design process is expected to concern the system in detail. Instead, one needs to use a top-down approach to the question [4]. A conflicting practice within architectural documentation today is that top-level architectural descriptions are not supported. Lower level documentation does not reflect all of the thoughts an architect had in mind in the early phase of the design. Documentation is important because, in most cases, the adapters of the architecture are not the creators. With high-level architectural descriptions available, it is easier for those adapting the architecture to use a top down method when familiarizing themselves with the structures and activities of a system.

It is also improbable that an architectural design process would not require iterations to optimize an architecture. These reasons mentioned above mean that a design method has to be divided into conceptual and concrete phases with an interface to requirements engineering (IFRE) above all, as shown in Figure 2. IFRE captures the technical properties and the context of the system. The conceptual architecture design phase models and documents the structure, behavior, deployment and development of the system at an abstract level. Concrete architecture defines the system structure, behavior, deployment and development in a more concrete sense using architectural descriptions produced in the conceptual design.

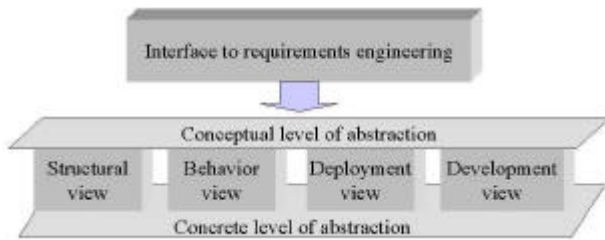


Figure 2. Views on two levels of abstraction in QAD.

The purpose of IFRE is to define the driving ideas of the system and identify the technical basis on which the system will be designed. Here, the driving ideas are considered as being the technical properties of the system, which equals with the main quality and/or functional goals this future system has to provide. In this phase a set of constraints are also defined. Constraints here mean specific standards or law issues (i.e. compliance) that constrict the development process of software architecture, e.g. protocol standards related to interoperability. In addition to the standards and laws, quality attributes may also restrict the design of architecture and should be taken into account in this early phase.

The conceptual software architecture provides organization of functionality and quality responsibilities into conceptual elements, collaboration between functional elements, and allocation of elements into hardware. These different aspects of conceptual software architecture are represented with four architectural views [10], [13], [15]: the structural view, behavior view, deployment view and development view.

The first view describes the structural viewpoint: software elements that compose the system, their interfaces and interconnections. Hierarchical structure is illustrated in a decomposition model, which is built up by clustering functional

responsibilities and mapped with the table of non-functional requirements (NFR).

The behavior view specifies the collaboration within a system; dynamic actions of and within a system, the kinds of actions the system produces and participates in, as well as their ordering and synchronization. The system behavior is described with a collaboration model.

The third view, the deployment view, clusters conceptual components into deployment units and describes the allocation of those units into physical computing devices. A table of units of deployment and a deployment model describe allowed allocations of units. The necessity of a unit in a system is presented in this view.

The conceptual development view is used to illustrate work allocation in component development or acquisition and it also captures the degree of reuse within the components. In the case of varying product features, the variability tree is represented.

The design rationale is a set of design principles and rules. Design rationale also provides the reasoning for why these principles and rules have been defined and possible consequences if they are neglected. Design rationale is related to an architectural description and can explain, for example, why a certain standard has been selected or the selected architectural styles with their preferences (Table 1).

Table 1. Styles, design rationale and quality attributes supported by styles

Style	Design rationale	Quality attributes
Layering	A layer provides coherent scope for modification caused by heterogeneous devices and communication manners.	portability, mobility, adaptability
Data oriented repository: blackboard	Data is distributed but must be considered as a block because of consistency requirement.	reliability, availability
Implicit invocation	The structure of a dynamic distributed system is non-stable. This style provides a means to manage continuously changing component states.	availability, extensibility
Object-oriented	The use of this style in service architectures is obvious but restricted. Support for reuse through generalization.	modifiability, adaptability

The concrete software architecture provides hierarchical containment of concrete software components and the defining of communication protocols used between them. The behavior of each component is described in detail, and finally, components are allocated to hardware resources, i.e. processors.

These different aspects of concrete software architecture are represented with four similarly named architecture views, as in the conceptual architecture: structural view, behavior view, deployment view and development view.

Architectural styles selected for the conceptual architecture are now complemented by design patterns [9]. These micro architectural elements [6] guide designers of components in greater detail when producing components and services compatible with the architecture. Appropriate design patterns are

recorded as design rationale of the concrete architecture. See an example in Table 2.

The first concrete view takes the decomposition model from the conceptual architecture as input and describes the structural viewpoint by means of refining components and interfaces between them. Hierarchical structure is illustrated in structure diagrams, which are built up in regard to refined non-functional requirements.

Table 2. Realizing styles through patterns

Pattern	Participates in the realization of a style	Used times in the system
Observer	Implicit invocation, Object-oriented, Blackboard	3

The concrete behavior view specifies the behavior of each component. The concrete system behavior is described with component state diagrams and message sequence charts.

The third concrete view creates software components that refer to concrete architectural components, i.e. to capsules and also to protocols. Software component instances that can be allocated to hardware processors illustrate this kind of deployment in a system level diagram.

The concrete development view defines how conceptual components are realized by concrete source modules for assembling and configuring software systems by retrieving components from an assets repository. The concrete development view links the architectural views to the assets management.

3. REALIZATION OF QUALITY ATTRIBUTES

Generally, the quality attributes of a software system are divided into two categories: functional and non-functional (NFR). The first one is the set of quality attributes that is observable at run-time (e.g. performance, functionality and usability). The second one is the set of quality attributes that cannot be discerned at the run-time, such as reusability or integrability [2].

The main quality attributes of software architectures are defined in [7]. Abbreviated definitions of those attributes are given in Table 3. As noticed, each of the quality attributes gives an overall quality definition for the whole system. Quality attributes at this level of abstraction cannot equal the system quality requirements as such. Instead, one needs to define the *scope* and requirement *definition* for each quality attribute involved in the system.

Quality *scope* means encapsulating a requirement or requirements as a responsibility of a *restricted part of the architecture*. The restricted part is instantiated e.g. as a component or a group of interacting components. The quality requirement *definition* means a more exact and system specific description of the quality attribute in question. For instance, the quality requirements for distributed service platform [16] architecture (for availability attribute), may be: (1) Service ‘A’ must always be active (and serving) in one of the distributed units and (2) the instances of the Service ‘A’ must be activated and shut down under control.

Under the interpretation above, the scoping and definition of quality attributes are the essential activities in realizing quality attributes. Furthermore, they support the selection and localization of *architectural styles and patterns*.

Architectural style is determined by a set of component types, a topological layout of the components, a set of semantic constraints and a set of connectors [2]. Patterns are widely reused and verified solutions for their specific problems and are grouped into three categories: architectural patterns, design patterns and idioms [6].

Architectural patterns express fundamental structural schema for software systems, which are applied in high-level system subdivisions, distribution, interaction and adaptation. A design pattern describes a recurring structure of communicating components, which solves a general design problem in a particular context [9]. Idioms are the lowest-level patterns. They describe how to implement particular aspects of components or relationships between them using the given language.

Table 3. Main quality attributes of software architectures

Attribute	Description
Performance	Responsiveness of the system.
Security	System’s ability to resist unauthorized attempts at usage and denial of service while still providing its service to legitimate users.
Availability	The proportion of time the system is up and running.
Reliability	The ability of the system or component to keep operating over the time or to perform its required functions under stated conditions for a specified period of time.
Usability	System’s learnability, efficiency, memorability, error avoidance, error handling and satisfaction concerning users’ actions.
Modifiability	The ability to make changes quickly and cost-effectively.
Maintainability	The ease with which a software system or component can be modified or adapt to a changed environment.
Flexibility	The ease with which a system or component can be modified for use in applications or an environment other than those for which it was specifically designed.
Scalability	The ease with which a system or component can be modified to fit the problem area.
Portability	The ability of the system to run under different computing systems: hardware, software or combination of the two.
Reusability	System’s structure or some of its components can be reused again in future applications.
Integrability	The ability to make the separately developed components of the system work correctly together.
Interoperability	The ability of a group of parts to exchange information and use the one exchanged.
Testability	The ease with which software can be made to demonstrate its faults.

A taxonomy of formally defined orthogonal properties of software architectures (TOPSA) [5] extends an architecture definition. It defines a space with three dimensions: abstraction (conceptual, realization), dynamism (static, dynamic) and aggregation. Static architecture is present in the code and can only change during development, whereas the dynamic architecture is the result of executing this code and can change once execution has started.

Our method defines four views on software architecture at two levels of abstraction. Each of these views and also the method’s interface to the previous development phase i.e. IFRE, concern the quality attributes defined in Table 4. As seen, the most

important quality attributes vary, depending on the current position on the abstraction axis of the TOPSA space.

In addition to quality attributes, the views also address *variability*, which is relevant for product line software architectures. Product line software architecture is a software architecture and a set of reusable components shared by a family of products [2], wherein variability denotes the differences among the product family members', i.e. products' architectures. High quality is essential in product line architectures because of their high degree of reuse.

Table 4. Mapping the QAD method to quality attributes

Method element	The most important quality attributes	
	Common	Special
IFRE		Interoperability, compliance
Conceptual Architectural Views		
Structural	Maintainability, modifiability	
Behavior		
Depl.		Performance, security, availability
Dev.		
Concrete Architectural Views		
Structural		Adaptability, portability
Behavior		Extensibility
Depl.		Interoperability, performance
Dev.		Intergrability

Variability appears on all orthogonal axis of the TOPSA space: abstraction, dynamism and aggregation. These dimensions of variability are addressed in four conceptual views of the QAD method as follows: At abstraction levels variability appears through inheriting (structural view, classes). At dynamism axis variability occurs both in static and dynamic dimensions of architecture (behavior view, functional scenarios and deployment view, allocation). At aggregation levels variability appears through component composition (development view).

Variability in concrete architecture can only be supported with frameworks that support product line approach, e.g. RoseRT [17] models. This way, an instance of a framework is the architecture of a single system.

4. CONCLUSION

In this paper, four viewpoints of software architecture at two abstraction levels have been defined. To fulfill the quality requirements that have been set for software product requires considering architectural viewpoints in the software development. Again, in order to reach quality attributes with architectural structures, the use of architectural styles and patterns is required.

In addition to quality requirements, software architecture has to respond to the functional demands formulated by the customer and end-users. Concentrating the attention on how to implement functional requirements in a concrete meaning is not a solution to this issue. Instead, a top down approach, i.e. an abstract viewpoint above concrete structures and activities is required.

Utilizing these constructs mentioned above, the QAD method provides an explicit and quality-driven link between software requirements and the realization of software architecture.

5. REFERENCES

- [1] Bachmann, F., Bass, L., Chastek, G., Donohoe, P. and Peruzzi, F. The Architecture Based Design Method. CMU/SEI, Technical report, 2000.
- [2] Bass, L., Clement, P. and Kazman, R. Software architecture in practice. Addison Wesley, Reading MA, 1998.
- [3] Bass, L., Klein, M. and Bachmann, F. Quality Attribute Primitives and the Attribute Driven Design Method in Proceedings of PFE-4 (Bilbao Spain, October 2001), 163 - 176.
- [4] Bosch. J. Design & Use of Software Architectures. Addison Wesley, Harlow, 2000.
- [5] Brathall, L. and Runeson, P. A Taxonomy of Orthogonal Properties of Software Architectures in Proceedings of NOSA'99 (Ronneby Sweden, August 1999).
- [6] Buschmann, F. Pattern-oriented software architecture, a system of patterns. John Wiley & Sons, Chichester, 1996.
- [7] Dobrica, L. and Niemelä, E. A Strategy for Analysing Product Line Software Architectures. VTT Publications 427, Technical Research Center of Finland, Espoo, 2000.
- [8] Dobrica, L. and Niemelä, E. Attribute-based product-line architecture development for embedded systems in Proceedings of AWSA'00 (Sydney Australia, Nov 2000), Monast University, 76 – 88.
- [9] Gamma, E. Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading MA, 1994.
- [10] Hofmeister, C., Nord, R. and Soni, D. Applied Software Architecture. Addison-Wesley, Reading MA, 2000.
- [11] IEEE Computer Society, IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems. IEEE Std-1471-2000.
- [12] IEEE Computer Society, Information Technology – Software Quality characteristics and metrics – Part 1: Quality characteristics and sub-characteristics. ISO/IEC 9126-1. 1996.
- [13] Jaaksi, A., Aalto, J-M., Aalto, A. and Vättö, K. Tried & True Object Development. Industry-Proven Approaches with UML. Cambridge University Press, New York, 1999.
- [14] Kang, K. C., Kim, S., Lee, J. and Kim, K. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5 (1998), 143-168.
- [15] Krutchen, P.B. The 4+1 View Model of Architecture. IEEE Software, 12 (November 1995), 42-50.
- [16] Matinlassi, M., Niemelä, E. and Dobrica L. Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. VTT Publications 456, Technical Research Center of Finland, Espoo, 2002.
- [17] Rational Rose RealTime CASE tool.
<http://www.rational.com/products/rosert/index.jsp>