Quality-Driven Architecture Design Method

Matinlassi Mari, Niemelä Eila
P.O. Box 1100, 90571 Oulu
Tel. +358 8 551 2111 Fax +358 8 551 2320
{Mari.Matinlassi, Eila.Niemela}@vtt.fi

Abstract: In this paper we introduce a quality-driven architecture design (QAD) method with three phases: system analysis, conceptual architecture design and design of a concrete architecture. Architecture design phases produce an aspect of software architecture at two different levels of abstraction. System analysis provides a bigger picture of the software system and its properties. Conceptual architecture organises these functional and quality responsibilities into conceptual elements, while a concrete architecture describes hierarchical components and communication protocols between them. Architectural descriptions at both abstraction levels are defined from three viewpoints: structural, behaviour and deployment. The QAD method has been applied with in a case study of a distributed service platform with the mobility of system services.

Key words: Software architecture, design methods, architectural descriptions, service architectures
1. Introduction

Architecture is the fundamental organisation of a software system embodied in its components, their relationships to each other and to the environment [2]. Software architecture also includes the principles guiding its design and evolution [10], and therefore, it has a strong influence over the life cycle of a system.

In the past, hardware engulfed other aspects of a system, and especially quality attributes like modifiability, interoperability and reusability were sacrificed first in the course of system development. Today, software-intensive systems are pervasive. 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 and the importance of quality-oriented design.

An architecture design method should cover the following questions:

- What are the entities represented and manipulated by the design method?
- What are the concrete means to describe architecture?
- What are the design steps and their ordering?
- How to apply the method?

Software architecture has to be described so that various stakeholders, e.g. customers, management and developers, understand it. In this light, it is obvious that one kind of architectural description is not enough, but the architecture has to be described with several different views. Furthermore, different views support analysis of particular quality attributes [17].

The FORM method [13] presents a solution for transformation of requirements to an architecture style. However, software systems need more than one style to follow. Architecture styles and design patterns are applied as driving factors in [2] but mapping requirements to software architecture is quite vague.

Architecture Based Design (ABD) method [1] is a quality driven method for designing the software architecture for a long-lived system at the conceptual level of abstraction. In ABD, the conceptual architecture [9] is a representation of the high-level design choices described with three architectural views. In spite of the ABD method has been developed further to 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. Also the support for product line architecture design in the ABD and in the ADD is mentioned but immature.

UML, with the strength of earlier widely applied object oriented methods [12, 19, 21] gives it benefits as a unified modelling language [20]. However, its superiority as an architectural description language is debatable.

This paper represents a quality-driven architecture design (QAD) method [17] that provides a systematic way to transform functional and quality requirements to software architecture. In discussion, we will summarise our experiences on the use of the method for designing architecture of the case study and outline our future directions.

2. Method

When designing software architectures it is not feasible to begin with the bottom-up style because it expects one to concern the system in details. Instead one needs to use a top-down approach to the issue [4]. Conflicting practice of the architectural documentation today is that it does not support high-level architectural descriptions. With high level architectural descriptions available it is easier for adapters of the architecture to use a top down method when getting familiar with the structures and activities of a system. It is also improbable that an architectural design process would not require iterations to optimise an architecture. These reasons mentioned above cause a design method to be divided into conceptual and concrete phases with system analysis above all, as shown in Figure 1.

![Figure 1. QAD method's main phases.](image-url)
System analysis captures the technical properties and the context of the system. Conceptual architecture design phase models and documents the structure, behaviour and deployment of the system at an abstract level. Concrete architecture defines system structure, behaviour and deployment in a more concrete sense using architectural descriptions produced in the conceptual design. Quality of both architectures, conceptual and concrete, is assessed by the architectural quality analysis [6, 7] and thereafter, required changes are updated to architectural models.

A method should give a series of steps to follow [14]. The detailed steps of this method are described in [17] and are not included here because of space limitations. For the same reason, the architecture quality analysis is out of the scope of this paper. In the next sections are described the overview of the method and the main phases with related architectural descriptions.

2.1 System analysis
The purpose of system analysis is to link the requirements engineering phase of the development life cycle and the software architectural design. Requirements engineering considered here, identifies the driving ideas of the system and the technical properties on which the system is to be designed. Detailed functional requirements are to be clustered in the conceptual architecture design phase of the method.

2.2 Conceptual architecture design
The conceptual software architecture [1, 9] provides organisation 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 three architecture views: structural view, behaviour view and deployment view. A view is defined to be a representation of a whole system from the perspective of a related set of concern [10]. Every view produces its specific architectural descriptions (Figure 2).

![Figure 2. Design phases and architectural descriptions of conceptual architecture.](image)

The first view describes the structural viewpoint: software elements that compose the system, their interfaces and interconnections. Hierarchical structure is illustrated in a logical model, which is built up by clustering functional responsibilities and mapped with the table of non-functional requirements (NFR).

The collaboration view specifies the behaviour of 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 synchronisation. The system behaviour is described with a collaboration model.

The third view, the deployment view, clusters conceptual components into deployment units and describes 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.

Design rationale is a set of design principles and rules. Design rationale also provides the reasoning 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.
2.2.1 Conceptual Structural View

The structural architecture encloses conceptual subsystem components, leaf components and their conceptual relationships (*Table 1*). In addition, variability points specified at the conceptual level and architectural styles used are illustrated in diagrams.

*Table 1.* Conceptual structural design elements, types and responsibilities.

<table>
<thead>
<tr>
<th>Element</th>
<th>Types of an element</th>
<th>Responsibilities of an element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual structural component</td>
<td>System component</td>
<td>What it has to do?</td>
</tr>
<tr>
<td></td>
<td>Subsystem component</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Leaf component</td>
<td>How it does it?</td>
</tr>
<tr>
<td>Conceptual structural relationship</td>
<td>Passes-data-to</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Passes-control-to</td>
<td>Why? (Design Rationale)</td>
</tr>
<tr>
<td></td>
<td>Uses</td>
<td></td>
</tr>
</tbody>
</table>

Conceptual elements divide the system into large functional blocks. At the top level, the system is decomposed into subsystems and at the next level, every conceptual subsystem is divided into a few conceptual components. At the end of this decomposing process, conceptual components are the smallest blocks used in this conceptual architecture level. Partitioning into subsystems and conceptual components has to be done with respect to the functional and quality aspects. Quality attributes drive the selection of architectural styles and thus affect the forming of the system architecture.

Conceptual components have conceptual relationships between each other. Relationships are abstracted interfaces between components, describing if an element passes control or data to, or has uses dependency with another component.

The responsibilities of the element define its role within the system. Responsibilities include both functional requirements and quality-oriented items and should answer the questions 'how' and 'why' besides the simple, common question 'what'.

2.2.2 Conceptual Behaviour View

The conceptual behaviour view is used to specify the behaviour of a system at a high abstraction level. Behaviour is recorded by defining conceptual behaviour elements: components and relationships and the responsibilities these elements have in the system (*Table 2*).

*Table 2.* Conceptual behaviour design elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Types of an element</th>
<th>Responsibilities of an element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual behaviour component</td>
<td>Service component</td>
<td>What it has to do?</td>
</tr>
<tr>
<td>Conceptual behaviour relationship</td>
<td>Ordered sequence of actions</td>
<td>How it does it?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Why? (Design Rationale)</td>
</tr>
</tbody>
</table>

Conceptual behaviour components i.e. service components are derived from conceptual structural components. Service component is equal either to a subsystem component or to a leaf component defined in the conceptual structural view. Selection depends on the size and complexity of the system under development.

Service components have behaviour relationships between each other. Behaviour relationships are ordered sequences of actions among a set of service components.

Naturally, the number of action sequences in a complex system is infinite and thereafter only the most essential sequences of actions are considered here. Essential sequences of actions are called collaboration scenarios and each collaboration scenario has a set of services related to it.

The collaboration model is an aggregate of collaboration diagrams. Each collaboration diagram represents one collaboration scenario graphically. In earlier design steps the service components were identified and essential collaboration scenarios selected from an infinite number of action sequences. In this final step the question 'in what order are actions done in each scenario' is answered and drawn into a set of collaboration diagrams.

2.2.3 Conceptual Deployment View

The deployment view is used to identify the distribution of hardware nodes in the system, group conceptual components to units of deployment and specify possible allocation of deployment units in computing units (*Table 3*).
The system hardware is described by means of distributed computing units called deployment nodes. Each deployment node is a platform for various services. Combination of services in different deployment nodes may vary and thereafter, conceptual leaf components are clustered into units of deployment.

A deployment unit is composed of one or more conceptual leaf components. Clustering is done according to mutual requirement relationships between components. In other words, a unit of deployment is atomic in the deployment process i.e. it cannot be split and deployed on more than one node.

<table>
<thead>
<tr>
<th>Element</th>
<th>Types of an element</th>
<th>Responsibilities of an element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment node</td>
<td>Various, depends on system</td>
<td>What it has to do?</td>
</tr>
<tr>
<td>Unit of deployment</td>
<td>Mandatory, alternative, optional</td>
<td>How it does it?</td>
</tr>
<tr>
<td>Conceptual deployment relationship</td>
<td>Is-allocated-to</td>
<td>Why? (Design Rationale)</td>
</tr>
</tbody>
</table>

Table 3. Conceptual deployment design elements.

Each unit of deployment represents one of three alternative types. These types are mandatory, alternative and optional [13, 18].

Allocation relationships are recognised between deployment nodes and units of deployment. The allocation relationship represents which services will be deployed in which distributed nodes/devices.

2.3 Concrete Architecture Design

Concrete software architecture refines conceptual structural components and their relationships. Also collaboration between components is described in more detail and detailed lower level components are allocated to hardware.

These different aspects of concrete software architecture are represented with three similarly named architecture views as in the conceptual level of abstraction: structural view, behaviour view and deployment view. In this concrete approach to architecture, every view produces its architectural descriptions (Figure 3).

Architectural styles selected for the conceptual architecture are now complemented by design patterns [8]. These micro architectural elements [5] guide designers of components in greater detail to carry out the components and services compatible with the architecture. Appropriate design patterns are recorded as design rationale of the concrete architecture.

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

The concrete behaviour view specifies the behaviour of each component. The concrete system behaviour 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 [22] 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 QAD method has been experimented in a case study of distributed service platform (DiSeP). The purpose of the DiSeP is to make software components in a networked environment interact spontaneously. In the DiSeP, the software components are various kinds of services that are either a part of the platform or a part of the application that utilises the platform. Interaction denotes that distributed parts of the platform or an application provide services, and/or use services that are provided by somebody else.

The configuration of the network may change dynamically. The main goal is to maintain the interoperability despite the dynamic nature of the network and the differences in the hardware or in the software implementation.

The QAD method is also under validation in two research projects with industrial case studies. The other one is an international project of wireless internet service engineering and the other case considers integration of COTS components to a service platform.

3. Discussions

We started by studying the ability of existing design methods [9, 11, 15] and drew the conclusion that simplicity, consistence and understandability are the key issues of most importance in a method applicable for the development of architectures. For simplicity, three viewpoints and two separate abstraction levels were defined. Three viewpoints were the minimum that was needed in the case study. However, development view for work allocation, third-party components and assets management is required in practice. Separation of abstraction levels makes a clear distinction between the issues described at the conceptual level and the matters considered in the concrete architecture development. In practice, the real problem is that the decisions that should be made at the component level are already defined in the conceptual architecture and therefore, architectural descriptions are confusing and difficult to understand.

Transformation of systems requirements to architecture and the use of the same viewpoints in the conceptual and concrete architecture assist in providing consistence between views. Traceability of systems properties is also possible due to mapping tables between the requirements and responsibilities of conceptual components. A recorded design rationale assists in understanding the comparisons, evaluations and decisions made during the development.

On the basis of the case study done with the QAD method, we see it as having several advantages. The method provides a systematic way to transform functional and quality requirements to software architecture and it also guides how to document the architecture. As a quality driven method, the QAD utilises architectural styles and patterns as a guide to carry out quality requirements in architectural descriptions. Quality-driven design provides that one defines the scope and requirement for each quality attribute involved in the system, because a quality attribute only gives an overall quality definition for the whole system. The scoping and definition of quality attributes are the essential activities in realizing quality attributes [16]. Furthermore, they support the selection and localization of architectural styles and patterns. QAD supports the product line architectures through documentation of variability and it is especially aimed at service architectures, which are considerably raising their necessity. In addition to these, the QAD method provides systematic progression steps, it is simple to learn and applicable to existing modelling tools.

Despite the fact that the QAD method has several advantages, there are still several issues to improve. Because the method is new, it has to be validated with several industrial case studies. Case studies should cover different kinds of service architectures, e.g. wireless services, value-added services and ubiquitous computing. Thus, new viewpoints have to be defined, especially for the stakeholders defined in business models but also for the varied end-users of ubiquitous computing systems. The deployment view, especially in service architectures, seems to have an importance in assisting the organisation of the design work between software architects and component designers but also between subcontractors and the persons responsible for acquisition of third-party components. On the lower abstraction level the development view defines how conceptual components are realised by concrete source modules, executables, libraries and make files 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.

Smooth linkage between the design and analysis of software architecture also needs further studies in order to provide a toolbox with a comprehensive set of analysis methods for all quality attributes. Further, the selection of architectural styles would be easier with a repository of styles supporting individual quality attributes. Design rationale is an equally important issue and the description of the design rationale must be unified through the design process.

Because QAD is the first version of the design method, the purpose of the method engineering was not to define an explicit method language. This is why experimental notations were used in addition to UML. However, strictly defined extensions for UML are needed and are under development.
4. Conclusion

In this paper, three viewpoints to architecture at two abstraction levels have been defined. Furthermore, suggestions on documenting these views with models and diagrams are introduced. To make these views and models useful, the method is experimented with upon existing tools in a service architecture case study and it is under evaluation in the WISE project\(^1\) and in a national joint research project with industrial case studies.

To fulfill the quality requirements that have been set to a 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 are required. This is done through refining and scoping the quality attributes. Furthermore, software quality analysis in the early design phases supports quality-driven design.

In addition to quality requirements, software architecture has to answer to the functional demands defined by the customers and end-users. Utilising the constructs mentioned above, the QAD method provides an explicit and quality-driven link between software requirements and architecture.

Acknowledgements

Finally, we would like to acknowledge the support provided by the PLA program.

References


\(^1\) A European project, IST-2000-30028.