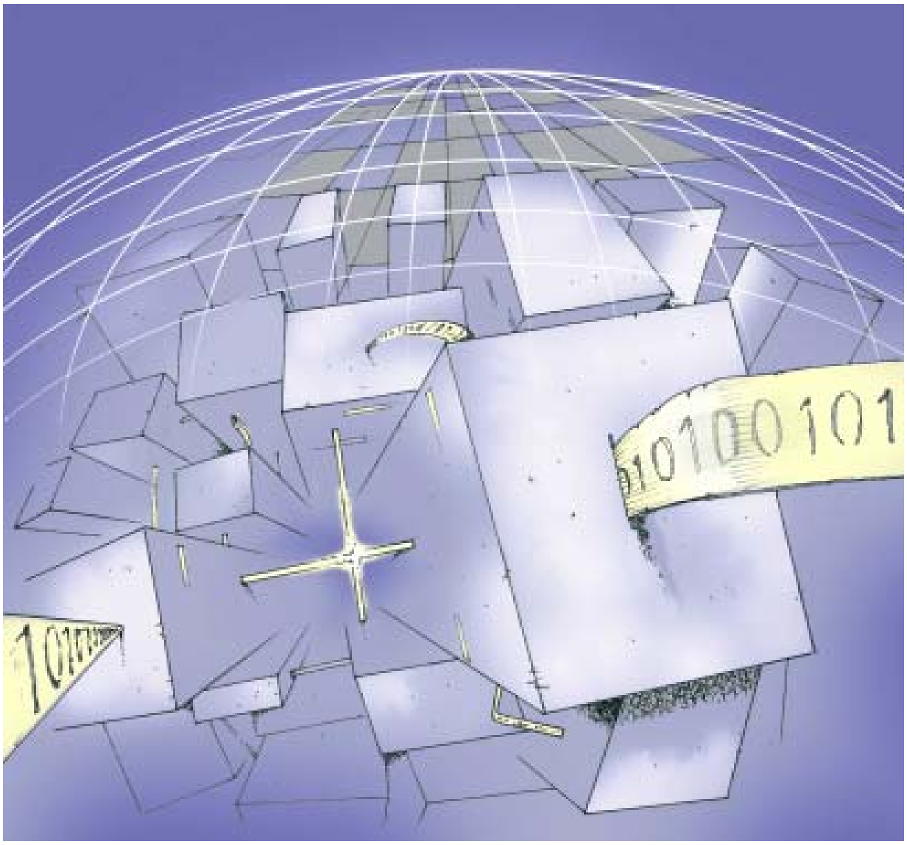
456

V T T   P U B L I C A T I O N S

Mari Matinlassi, Eila Niemelä & Liliana Dobrica

Quality-driven architecture design and quality analysis method

A revolutionary initiation approach to a product line architecture



TECHNICAL RESEARCH CENTRE OF FINLAND   ESPOO 2002

VTT PUBLICATIONS 456

Quality-driven architecture design and quality analysis method

A revolutionary initiation approach to a product line architecture

Mari Matinlassi

Eila Niemelä

Liliana Dobrica

VTT Electronics

TECHNICAL RESEARCH CENTRE OF FINLAND

ESPOO 2002

ISBN 951–38–5967–3 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–5968–1 (URL:http://www.inf.vtt.fi/pdf/)
ISSN 1455–0857 (URL:http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Otamedia Oy, Espoo 2002

Matinlassi, Mari, Niemelä, Eila & Dobrica, Liliana. Quality-driven architecture design and quality analysis method. A revolutionary initiation approach to a product line architecture. Espoo 2002. Technical Research Centre of Finland, VTT Publications 456. 128 p. + app. 10 p.

Keywords software product line, architecture design and analysis methods, quality attributes, service architecture, software architecture, QADA

Abstract

The role of software architecture has changed. The use of modern software technologies and practices enables turning the focus of system development to the quality aspects of software instead of functional properties. Architecture addresses the quality issues of software and, therefore, it must be developed and documented properly. In particular, there is a need for high level architectural descriptions. The top down nature of software architecture design induces this need.

In this report we introduce a quality-driven architecture design and analysis (QADA) method. Quality-driven is about *utilizing architectural styles and patterns* as a means of designing high-quality architectures. QADA takes a revolutionary approach to the initiation process of a new product line. That is, the development of a complete product-line architecture and a set of components before developing the first product in a new domain. QADA considers architecture on two levels of abstraction: *conceptual* and *concrete*. *Design* produces architectural descriptions at both abstraction levels from three viewpoints: *structural*, *behavior* and *deployment*. The structural viewpoint is concerned with composition of software components, whereas the behavior viewpoint takes the behavioral architecture aspects under consideration. The deployment viewpoint refers to embedding and allocation of software components to various computing environments. Quality of architecture on both levels of abstraction is *analyzed* in the corresponding analysis phases.

Because software architectural design is difficult to discuss merely at an abstract level, the QADA method is tested with a case study of a distributed service platform. The platform embodies a layered service architecture, thereby providing a variety of services for its users. The upper layer of services, i.e. the system services of the platform is mobile, enabling spontaneous networking.

Preface

The research discussed in this report was carried out as a result of the PLANA project during the year 2001 at VTT Electronics. PLANA, Product Line Architecture Design and Analysis Methods, is a part of the PLA programme. The PLA (Product Line Architectures) programme is a self-funded strategic research programme of VTT Electronics. The aim of the PLA programme is to create and intensify the knowledge of software architectures and assets management required in the development of product lines. PLA design and analysis methods and techniques, as well as architectural concepts for secure and mobile applications, are the key issues.

The driving forces behind the development of a product line architecture are reusability and modifiability. An essential question is which architectural views and descriptions are needed, for whom and how long they have to be maintained. The open problem is how to take better advantage of architectural concepts to analyze a software product line for quality attributes in a systematic way. It is also important to estimate how reusable the product line architecture is in relation to anticipated changes and verify that the quality requirements of the domain have been addressed in the product line architecture design.

The main objectives of the PLANA project are to develop a design method for product line architectures, to develop analysis methods for product line architectures of middleware and system infrastructure services, and create new national and international co-operation in product line architecture research.

The results of this study have been carried out in close co-operation between the equal co-authors. Mari Matinlassi has studied the practical case study according to the guidance of Eila Niemelä. The development of the design method has required the work and experience of both. Liliana Dobrica has mostly developed the analysis method. However, Eila Niemelä and Mari Matinlassi have contributed to her work by providing the architectural descriptions to be analyzed and also commented on the documents produced by Liliana Dobrica.

Oulu, Finland, January 2002

Mari Matinlassi Eila Niemelä Liliana Dobrica

Contents

Abbreviations

API	Application Programming Interface
AQA	Architecture Quality Analysis
CASE	Computer Aided Software Engineering
COMET	Concurrent Object Modeling and Architectural Design with UML
CSV	Commonality, Scope and Variability analysis
CVA	Customer Value Analysis
DiSeP	Distributed Service Platform
DTB	Data Transfer Block
IEEE	Institute of Electrical and Electronic Engineers
ISO/IEC	International Standardization Organization and International Electrotechnical Commission
FORM	Feature Oriented Reuse Method
MSC	Message Sequence Chart
NFR	Non-functional Requirements
PL	Product Line
PLA	Product Line Architecture
QAD	Quality-Driven Architecture Design
QADA	Quality-driven Architecture Design and quality Analysis

QFD	Quality Function Deployment
SAAM	Scenario-based Architecture Analysis Method
UML	Unified Modeling Language
XML	eXtended Markup Language

1. Introduction

Software architecture is the fundamental organization of software embodied in its components, their relationships to each other and to the computing environment [28]. 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.

In the past, hardware engulfed the 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. 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.

Software architecture can be considered as a *reusable element* solving the problem of how to bridge the gap between requirements and code. The process of filling this gap requires one to consider the two major relationships between these three elements. Firstly, how to turn the requirements into the form of an architecture, and secondly, how again to turn the architecture into implementations.

The FORM method [35] presents a solution for the transformation of requirements into an architecture style. However, software systems need more than one style to follow. Architecture styles and design patterns are applied as driving factors in [9] but mapping requirements for the software architecture are quite vague. UML with the strength of earlier widely applied object oriented methods [33, 54, 56] gives it benefits as a unified modeling language [55]. However, its superiority as an architectural description language is debatable.

The COMET method [24] concerns architectural design as a part of the application design. Designing the system architecture is the third and final phase of the method. The first phase is requirements modeling with use cases, and the second one is analysis modeling, which contains developing the static and dynamic class and object models. However, software architecture design cannot be considered as a phase in a design process started with use cases. Quality oriented items i.e. styles and patterns have to be considered as very important when building architectures, because they act as drivers in the selection of architectural structures and act with the whole design process, from the requirement specification down to the implementation and product maintenance.

Some of published architectural design methods concentrate on different *views* of architecture. Inside this group there is a great number of different views and ways to describe the architectural documentation of views in use. The first of these view-oriented methods was the 4+1 developed by Krutchen [38]. After this, several others have approached the zoo of architectural viewpoints. For instance, Jaaksi et al. introduced their 3+1 method in 1999 [32] and Hofmeister et al. used all in four views to describe architecture [27].

It is obvious that none of these methods alone is comprehensive enough to cover the *design* of software architectures for systems of a different size on various domains, or provide an explicit means to create *architectural descriptions* for these systems. Instead, the need for different architectural views and architectural documents is highly dependent on the two issues: system size and software domain e.g. the application domain, middleware service domain and infrastructure service domain.

Design and use of *software architectures* are closely related to software quality, e.g. reliability, performance and modifiability. *A software product line* consists of a product-line architecture and a set of reusable components that are designed for incorporation into the product-line architecture [12]. In addition to the quality that is achieved by adopting a quality-driven architecture design, there are additional reasons to adopt the product-line approach, namely decreased development and maintenance costs and shorter time-to-market.

Some of the issues with a software product line are related to the process of initiation and how to deal with its evolution process. Considering the initiation, a software product-line does not appear accidentally, but requires a conscious and explicit effort from the organization interested in using the product-line approach. Basically, one can identify two relevant dimensions with respect to the initiation process. The organization may take an evolutionary or a revolutionary approach to the initiation process. In each dimension the product-line initiation can be applied to an existing line of products or to a new product-line that the organization intends to use. Each case has an associated risk level and benefits.

For instance, in general, the revolutionary approach involves more risk, but higher returns compared to the evolutionary approach. The revolutionary approach to a new product-line means that product-line architecture and components are developed to match the requirements of all expected product line members, before developing the first product in a new domain.

There is an opinion that considers that product-line architectures should differ from single product architectures in that they provide solutions specific to the product members that build the domain [26]. In this sense, architectural modeling for product lines should go beyond defining simple connectors and components for the architectural views employed.

It is as important to check the architecture against quality requirements in order to achieve quality attributes, as is the design of architecture. Checking is about *analyzing* and *evaluating* the architecture. The aim of *analyzing* the architecture is to predict the quality of a software system before it has been built, and not to establish precise estimates about the principal effects of the architecture. In *evaluation* the architecture is compared with another architectural candidate or with requirements set for products, in order to minimize risks and prove that requirements have been addressed in the design. Instead, *an analysis* brings out opinions whether the architecture is proper and what are its possible weaknesses. Opinions are based on evaluation results.

The research in the PLA domain is at the outset, because software product-line architecture has came into the scene recently. In the case of single software products architecture analysis methods are mature enough and the most representative have been presented and compared in a survey [20]. However, when considering the analysis methods of software product-line architecture the work in [18], where a strategy for analyzing product-line architectures is introduced should be mentioned.

This report introduces a quality-driven architecture design and quality analysis (QADA) method that provides a systematic way to transform functional and quality requirements into software architecture. The method also utilizes styles and patterns as a guide to carry out quality requirements in architectural descriptions with a documented design rationale. The use of the QADA method is adapted for a commercial CASE tool, RoseRT [51], and demonstrated with a case study. The case study concerns the revolutionary initiation of a new product line of distributed service platforms in spontaneous networking systems. Revolutionary initiation of a new product line means the development of a complete product-line architecture and set of components before developing the first product in the new domain [12]. At the end, we will summarize our experiences on the use of the method for designing and analyzing the architecture of the case study.

2. Background

This section defines the main terminology related to the QADA method. It also introduces concepts that are linked to quality and quality analysis of software architectures and presents a comparison of three architectural design methods based on architectural views.

2.1 Main terminology

A *method* is a description of how to conduct a process [37]. A process is an activity which takes place over time and which has a precise aim regarding the result to be achieved. The method description defines and organizes a collection of techniques and a set of rules that establishes how to conduct an activity. The set of rules of the method states *by whom, in what order, and in what way the techniques* are used to accomplish the method objective.

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.

Software architecture is defined as the structure or structures of the system, which consists of software components, the externally visible properties of those components and the relationships among them [9]. Software architecture also includes the principles and guides that control the design and evolution in time [49, 58].

Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only [60].

Software product line is a group of products sharing a common, managed set of features that satisfies the specific needs of a selected market [9]. Software products are instances of the software product line.

Revolutionary initiation approach of a new product line means development of a complete product-line architecture and set of components before developing the first product in the new domain [12].

Product-line architecture (PLA) is adaptable architecture that is applied to the product members of a product line and from which the software architecture of each product member can be derived. PLA is software architecture and a set of reusable components shared by a family of products [9].

Architectural view is a representation of a whole system from the perspective of a related set of concerns [28].

Viewpoint is a specification of the conventions for constructing and using an architectural view. A pattern or a template used to develop individual views, by establishing the purposes and audience for a view, and the techniques for its creation and analysis [28].

Architecture style defines a class of architectures and is an abstraction for a set of architectures that meet it. A 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 [9].

Architecture pattern expresses fundamental structural schema for software systems, which are applied for high-level system subdivisions, distribution, interaction and adaptation [15]. When the schema is strictly described and commonly available, it is a pattern.

Design pattern describes a recurring structure of communicating components, which solves a general design problem in a particular context [23]. Design patterns are micro architectures and do not guarantee a good overall architecture.

Mandatory feature is a feature that must always be included in a product of a product family [46, 35].

Optional feature is a specific feature that is contained in one product of product line, but not in another [4, 46, 35].

The architecture provides a placeholder in which one of several alternatives can be inserted. *Alternative feature* cannot coexist with other alternative features [4, 46, 35].

Middleware is software that is located between applications and the network layer, and is independent of operating systems. Middleware hides the distribution from applications.

Service is the capability of an entity (the server) to perform, upon the request of another entity (the client), an act that can be perceived and exploited by the client.

Service architecture is the architecture of applications and middleware. It is a set of concepts and principles for the specification, design, implementation and management of software services [61].

Customer value analysis seeks to quantify qualities that affect a customer's decision to buy a particular product. Below, the term *value* denotes the product's perceived overall benefit relative to its cost [21].

2.2 Quality attributes and quality model

A quality attribute is a non-functional characteristic of a component or a system, such as integrability, modifiability, reliability or availability. A software quality is defined in IEEE 1061 [29] and it represents the degree to which software possesses a desired combination of attributes. Another standard, ISO/IEC Draft 9126-1 [31], defines a software quality model. According to this model, there are six categories of characteristics (functionality, reliability, usability, efficiency, maintainability and portability) which are further divided into sub-characteristics. These are defined by means of externally observable features for each software system. In order to ensure its general application, the standard does not determine what these attributes are, nor how they can be related to the sub-characteristics.

An investigation into the literature has shown that a large number of definitions of quality attributes exist that are related to similar abilities of a software system. Quality attributes are defined in [18] and, for example, the definitions for maintainability, flexibility and modifiability are:

Maintainability is a set of attributes that have a bearing on the effort needed to make specified modifications [31]. Modifications may include corrections, improvements or adaptations of software to changes in environment, and in requirements and functional specification.

Modifiability is the ability to make changes quickly and cost-effectively [9]. Modifications to a system can be categorized as extensibility (the ability to acquire new features), deleting unwanted capabilities (to simplify the functionality of an existing application) portability (adapting to new operating environments) or restructuring (rationalizing system services, modularizing, creating reusable components).

Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [30].

Flexibility is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [30].

Although different in wording, the definitions are almost identical in their semantics. The limitation of these definitions with respect to the purpose of analyzing software architectures is that their scope is too broad. The scope has to be narrowed, based on the relevant context.

2.3 Design methods based on architectural views

Architectural views have been the basis for a number of design methods during the last few years. The first of them was the method of 4+1 views to software architecture [38]. The four main views used in this method are *logical, process, physical* and *development* view. The *logical* view describes an object model. The *process* view describes the design's concurrency and synchronization aspect. The *physical* view describes the mapping of the software onto the hardware reflecting the distributed aspect of the system. The *development* view describes the software's static organization in its development environment. The '+1' view is a use-case view consisting of *scenarios* that are used to illustrate the four views.

In [32], a slightly modified version of the 4+1 method is suggested and ends up with the 3+1 views necessary to describe the software architecture. The views in the 3+1 method are the *logical*, *runtime* and *development* view, plus the *scenario* view. The *logical* view illustrates the high level partitioning of the system into application products and applications. The *run-time* view specifies all the executable components of the system, while the *development* view specifies the components that are developed independently. The *scenario* view illustrates the collaboration of these components in different usage situations. The 3+1 method applies the Unified Modeling Language (UML) as an architectural description language.

In addition to the methods introduced above, [27] defines four views that are the *conceptual*, *module*, *execution* and *code* view. *Conceptual* view describes the system in terms of its major design elements and the relationships among them. *Execution* view addresses the dynamic structural issues e.g. allocation of software in distributed systems and synchronization aspects on architectural level. The main purpose of the *module* view is the decomposition of the system to modules and the partitioning of software modules into layers. The *code* view provides the organization of the source code into object code, libraries and binaries. The four views are based on observations done in practice on various domains e.g. image and signal processing systems, a real-time operating system, communication systems etc.

Table 1 illustrates the similarities between the views that are defined in the three methods introduced above. The method of the four views [27] is defined in great detail and is therefore used as the bases for the comparison. The artifacts of the four-view method and their architectural descriptions are introduced briefly in the columns on the left. The columns on the right describe the view defined in other methods (the 4+1 and the 3+1) that best corresponds to the description of the view.

Table 1 uses the first letter or two first letters of each view as an abbreviation of the view, e.g. C for the conceptual view, Ph for the physical view and Pr for the process view.

Table 1. A comparison between the views defined in the three methods.

The four views		4+1	3+1
The view and its descriptions	**Architectural artifacts**		
C. Conceptual configuration	UML Class Diagram	*L.*	*L.*
C. Port or role protocol	ROOM protocol declaration		
C. Component or connector behavior	Natural language description or UML State Diagram	*L.*	
C. Interactions among components	UML Sequence Diagram		
M. Conceptual-*module* view correspondence	Table	*D.*	
M. Subsystem and module decomposition	UML Class Diagram	*D.*	*D.*
M. Module use-dependencies	UML Class Diagram		
M. Layer use-dependencies, modules assigned to layers	UML Class Diagram	*D.*	*D.*
M. Summary of module relations	Table		
E. Execution configuration	UML Class Diagram	*Pr.&D.*	*R.&D.*
E. Execution configuration mapped to hardware devices	UML Deployment Diagram	*Ph.*	*R.*
E. Dynamic behavior of configuration, or transition between configurations	UML Sequence Diagram		
E. Description of runtime entities (host type, replication and assigned modules)	Table or UML Class Diagram		*R.*
Code. Description of components in code architecture view, their organization and their dependencies	UML Component Diagrams or tables	*D.*	*D.*

In addition to the artifacts mentioned in Table 1, the *execution view* provides two artifacts that are not supported by the 4+1 or 3+1 methods:

- Description of runtime instances (including resource allocation) in the form of a table.

- Communication protocols described with a natural language description, a UML Sequence Diagram or State Diagram.

Above and beyond, the *code view* also considers the following artifacts that are not mentioned in Table 1 and are not supported by the 4+1 or 3+1 methods:

- Module view, source component correspondence i.e. tracing dependency that is described with tables.

- Runtime entity, executable correspondence i.e. instantiation dependency that is described with tables.

- Descriptions of build procedures described with tool-specific representations (for example, make-files).

- Description of release schedules for modules and corresponding component versions that are described with tables.

- Configuration management views for developers described with tool-specific representations.

As descerned from Table 1, the method of 3+1 architectural views does not consider the *behavior of components* at an architectural level. Instead, the *behavior design* is part of the *detailed design*. The scenarios in +1 view are used to describe the behavior at the architectural level.

Despite the fact these methods mentioned above are capable and exhaustive in their own way, none of them concerns the product line approach to the architectural design.

2.4 Analysis at the architecture level

A classification of the analysis techniques available at the architecture level is defined in two important research reports [1, 10]. Here it is sufficient can be to identify two basic classes of techniques, *questioning* and *measuring*. The first class, consisting of *questioning techniques*, generates qualitative questions to be asked of an architecture, and it can be applied to evaluate an architecture for any given quality. This class includes scenarios, questionnaires and checklists.

Measuring techniques, that represent the second class, suggest quantitative measurements to be made on an architecture. They are used to answer specific questions and address specific software qualities and, therefore, they are not as broadly applicable as questioning techniques. This class includes metrics, simulations, prototypes and experiences. Metrics are quantitative interpretations that are placed on a particular observable measurement of the architecture, such as fan in/fan out of the components.

Generality, level-of-detail and the design phase may define a framework of comparison for possible evaluation techniques. Regarding generality, the techniques could be general (questionnaire), domain-specific (checklists, prototype) or system-specific (scenarios). The level of detail (coarse-grained, medium or fine) indicates how much information about the architecture is required to perform the evaluation. There are three phases of interest in architecture evaluation: early, middle and post-deployment. The early phase evaluation occurs after the initial high-level architectural decisions (questionnaire, prototype), the middle phase occurs at any point after some elaboration of the architecture design (scenarios, checklists), and post-deployment occurs after the system has been completely designed, implemented and deployed. During the last stage, both the architecture and the implementation exist and an evaluation of whether the architecture matches the implementation can be done.

The essential rules for analyzing software architecture to determine if it exhibits certain quality attributes are described in [7]. These rules provide a *context* for existing evaluation techniques. One of the first main rules is the identification of the contract between the system and the environment. Conforming to this rule, scenarios represent a form of expressing the expectations and the obligations of the system, and this technique defines what needs to be confirmed by the analysis.

In terms of quantitative and qualitative aspects, both classes of techniques are needed for evaluating architectures. Various analyzing models expressed in formal methods are included in quantitative techniques. Qualitative techniques illustrate software architecture evaluations using scenarios. A description of the changes that are needed for each scenario represents a qualitative method of evaluation. From this perspective, scenarios are rough, qualitative evaluations of architecture. Scenarios are necessary but not sufficient to predict and control quality attributes, and they have to be supplemented with other evaluation techniques, and particularly quantitative interpretations. For example, including questions about quality indicators in the scenarios enriches the architecture evaluation. Quantitative interpretations of scenario evaluations could be: ranking between the effects of scenarios i.e. a five level scale, or an absolute statement, which estimates the size of modifications or different metrics, such as lines of code, function points or object points.

2.5 Scenarios

Most of the architecture analysis methods use scenarios. The usage of scenarios is motivated by the consensus it brings to understanding what a particular software quality really means. Scenarios are a good way of synthesizing individual interpretations of a software quality into a common view. This view is more concrete than the general definition of software quality [30], and it also incorporates the specifics of a system to be developed, i.e. it is more context-sensitive.

Scenarios are a postulated set of use situations or modifications of the system. In analyzing a system, it is important that all stakeholders relevant to that system (operator, system designer, modifier, system administrator and others depending on the domain) are considered, since design decisions may be made to accommodate any of these stakeholders. Scenarios are typically one sentence long and could be more appropriately called *vignettes*. The modifications reflected in scenarios could be:

- a change as to how one or more components perform an assigned activity,
- the addition of a component to perform some activity,
- the addition of a connection between existing components, or
- a combination of these factors.

The development of scenarios is based on the system requirements that are reflected in the architecture. Scenarios have to be sufficiently concrete to ensure the expressiveness of the analysis. In this regard, it was demonstrated that it does not seem to be possible to assess the reusability of an architecture in general. That is, to depict typical reuse situations for applications in a respective domain in a set of scenarios [41]. Concentrating on a specific set of applications and specific reuse scenarios allows the eliciting of information on the flexibility of software architecture and its constraints.

In creating and organizing scenarios, the concept of involved organization or *stakeholder* related to the system is defined. Examples of such organizations include the following:

- The organization responsible for executing the software: the end user;
- The organization responsible for managing the data repositories used by the system: the system administrator;
- The organization responsible for modifying the runtime functions of the system: the developer;
- The organization responsible for approving new requirements for the system.

Each stakeholder may express a set of quality goals for the system.

The terminology and concepts presented earlier form a terminological framework for the discussion of the quality-driven architecture design and quality analysis method that will be presented in the next chapter.

3. Overview of the QADA method

When designing software architectures it is not feasible to begin with the bottom-up style, because it expects one to consider the system in detail. Instead one needs to use a top-down approach to the issue [12]. A conflicting practice of architectural documentation today is that it does not support top-level architectural descriptions. Lower level documentation does not reflect all the thoughts the architect had in mind in the early phase of the design. Documentation is important because, in most cases, the adapter of the architecture, e.g. a software integrator or administrator, are not its creators. With high level architectural descriptions available it is easier for the adapters of the architecture to use a top down approach when getting familiar with the structures and activities in a system.

The QADA method includes architectural design and quality analysis. Design and analysis are closely associated and appear in turn in architecture development (Figure 1). Architecture design is divided into conceptual and concrete levels of abstraction in order to create high architectural descriptions and analyze architecture quality also in the early phase of the design. Architecture analysis measures the quality of software architecture with scenario-based analysis methods and evaluates the architecture candidates by comparing them or comparing a candidate against the quality attributes set in the requirements engineering.

Requirements engineering captures and analyzes the technical properties and the context of the system. *Conceptual architecture design* [5] phase models and documents the structure, behavior and deployment of the system at an abstract level. *Conceptual quality analysis* is firstly about *analyzing* an architecture candidate (i.e. styles and patterns) against quality attributes that are relevant at this abstract level and secondly *evaluating* architecture candidates. *The concrete architecture design* phase defines system structure, behavior and deployment in more detail using the architectural descriptions produced in conceptual design as input. *Quality of the concrete architecture* is assessed again by the architectural quality analysis [18, 19] and evaluation and thereafter, the needed changes are updated to architectural models.

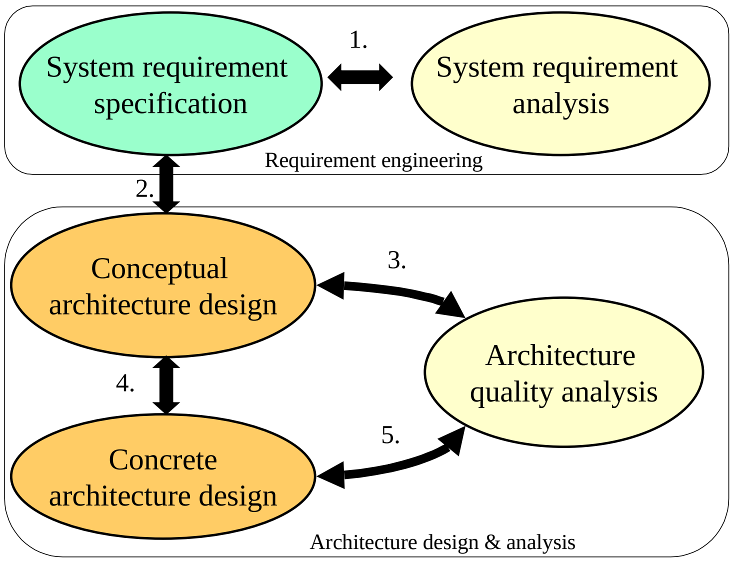

System requirement specification
1.
System requirement analysis
Requirement engineering
2.
Conceptual architecture design
3.
Architecture quality analysis
4.
Concrete architecture design
5.
Architecture design & analysis

Figure 1. QADA method main phases.

The QADA method considers that the architecture should be described with multiple views (Figure 2). During design and analysis, architecture views evolve from the conceptual level description to a more concrete level. A quality-driven design and analysis method may involve views which are concerned with:

- The decomposition, in a structural view, of the functionality that the products need to support. At a conceptual level this view is useful for understanding the interactions between entities in the problem space, planning functionality and understanding the domain *variability* and, hence thereafter, the possibilities of initiating a product-line. At a concrete level the elements from which the system is built could be crucial for understanding the maintainability, modifiability, reusability and portability of the system [cf. 18].

- Behavior view that is important in order to understand *performance* but, also *reliability* and *security*.

- Deployment view, including central processing units, memory, buses, networks and input/output devices. Quality attributes relevant to this view include *availability*, *capacity* and *bandwidth* as well as *performance*, *reliability* and *security*.

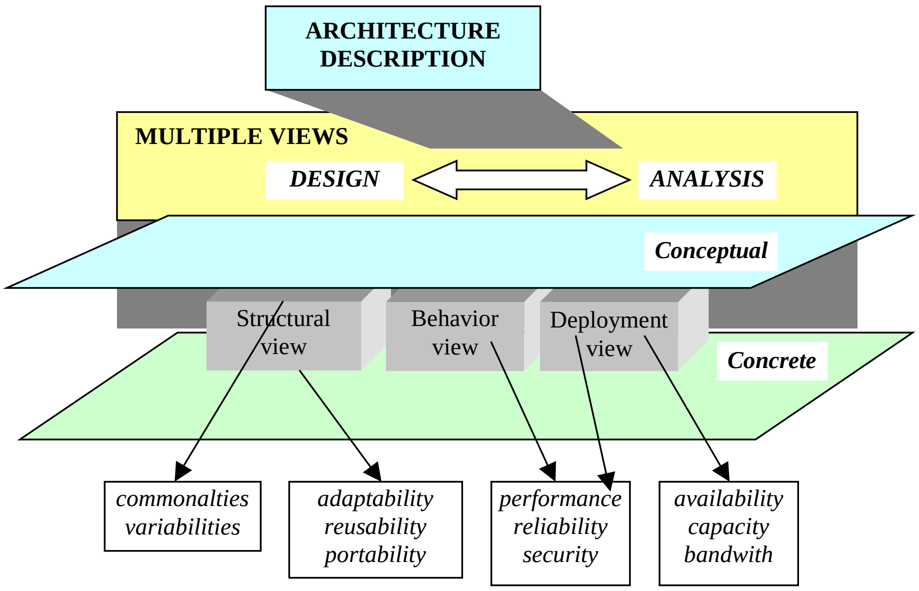

ARCHITECTURE DESCRIPTION
MULTIPLE VIEWS
DESIGN
ANALYSIS
Conceptual
Structural view
Behavior view
Deployment view
Concrete
commonalties variabilities
adaptability reusability portability
performance reliability security
availability capacity bandwith

Figure 2. Quality-driven architecture design and analysis.

Maintainability is a quality attribute related to the structural view, the behavior view and the *development vie.* The last is not considered here.

3.1 Requirements engineering

Requirements engineering considered here is an interface between the requirements engineering and architectural design. In requirements engineering the *technical properties* and the *context* of the system are defined. An important issue is also to *analyze the meaning of technical properties* for architecture design and *scope product line* of the system.

Technical properties are *functional and non-functional requirements* described by a textual list of properties. The context model defines the hardware and software interfaces of the system and a set of constraints, in the form of standards, rules set by laws, or quality requirements.

Figure 3 shows the architectural descriptions of the requirements engineering phase.

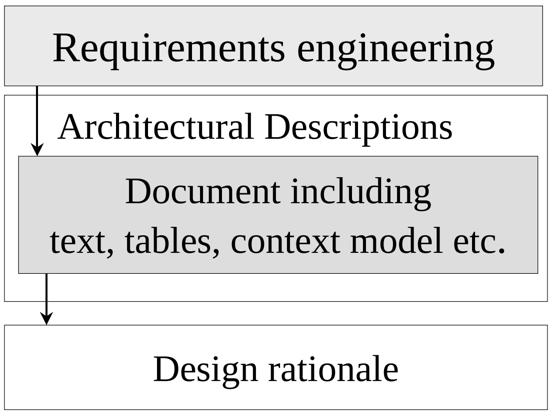
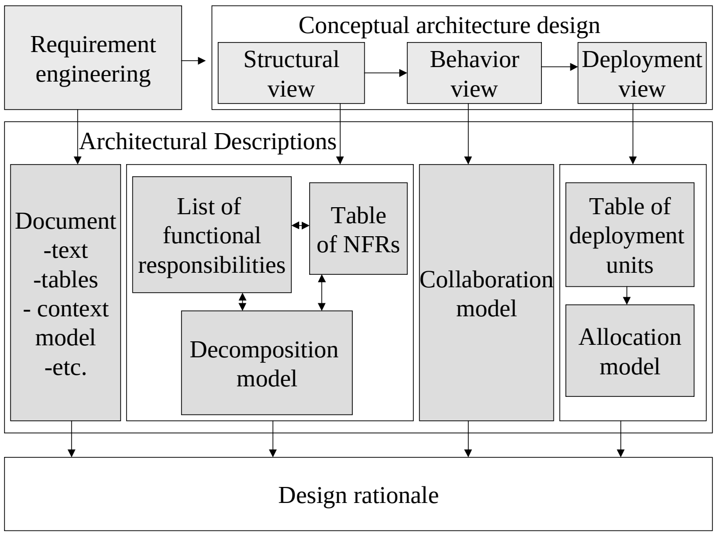


Figure 3. QADA method's requirements engineering phase.

3.2 Conceptual architecture design

The conceptual software architecture provides the organization of functionality and quality responsibilities into conceptual elements i.e. components and their relationships, collaboration between components and allocation of components to hardware.

The different aspects of conceptual software architecture are represented with three architecture views [27, 32, 38]: structural view, behavior view and deployment view, which are described more detailed in sections 5.1, 5.2 and 5.3. A view is defined to be a representation of a whole system from the perspective of a related set of concerns [28]. Every view produces its specific architectural descriptions forming the main part of the documentation of the architecture (Figure 4).



Requirement engineering
Conceptual architecture design
Structural view
Behavior view
Deployment view
Architectural Descriptions
Document -text -tables - context model -etc.
List of functional responsibilities
Table of NFRs
Decomposition model
Collaboration model
Table of deployment units
Allocation model
Design rationale

Figure 4. Design phases and architectural descriptions of the conceptual architecture.

The first view describes the *structural* viewpoint: software components that compose the system and their relationships. Hierarchical structure is illustrated in a decomposition model, which is built up by clustering functional responsibilities and selecting architectural styles addressed to certain selected non-functional requirements (NFR).

The *behavior* view specifies the system from the viewpoint of dynamic actions of, and within, a system, the kinds of actions the system produces and also participates in 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 allocation of those units into physical computing nodes. A table of units of deployment and an allocation model describe the allowed allocations of units. The necessity for 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 describe the selected architectural styles with their preferences.

3.3 Conceptual architecture analysis

In a product-line context, each software product member possesses a desired combination of quality attributes that define domain quality model. The quality model of a product-line domain must include the variability and commonality among the quality requirements and the architecture model should conform to these aspects. Product-line architecture includes commonality and variability indicating what can be common and different among the product members of a product line, respectively.

Conceptual product-line architecture shows the results of the earliest design decisions about a software system. Taking good decisions could lead to reduce costs and risks.

Conceptual architecture analysis focuses on becoming aware of the available and required information to do the analysis, and then to collect and compile it. Currently three categories of information are being addressed: *stakeholder, architecture and quality*. In the future, the information categories may be extended. This information is gathered in a knowledge base.

The first phase of product-line architecture analysis method in a revolutionary initiation approach is described in Figure 5. The product-line requirements define not only the product-line scope, but also represent the analysis input to create a knowledge base of the requirements’ taxonomy.

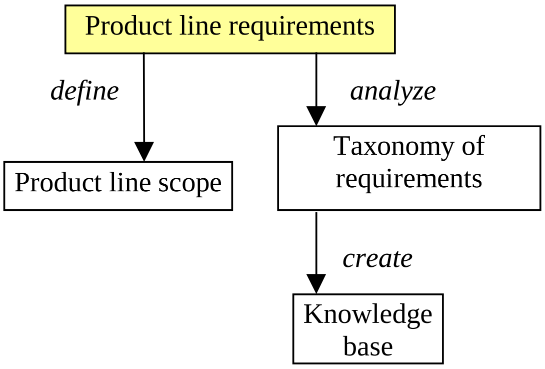


Figure 5. First phase of conceptual architecture analysis.

Syntactic architectural notations should be well understood by the parties involved in the analysis. The result of an architecture evaluation process depends on how well the description is made. This phase focuses on specific software architecture analysis and the generation of artifacts to do the analysis. Examples of artifacts include: domain models (which help in comparing competing architectures within the same functional area); relevant architectural views; architectural styles; environmental assumptions and constraints; and trade-off rationale.

The role of a knowledge base is to allow the evaluation of collections of architecture styles and patterns in terms of both quality factors and concerns, and anticipation of their use. A ”pre-score” of architectural patterns is feasible in order to gain a sense of their relative suitability to meet the particular quality requirements of a system. In addition to evaluating individual patterns, it is necessary to evaluate compositions of patterns that might be used in the architecture. Identifying patterns that do not compose well (the result is difficult to analyze, or the quality factors of the result are in conflict with each other) should steer a designer away from “difficult” architectures towards those made of well-behaving compositions of patterns.

The knowledge base built in this way helps to move from the notion of architectural styles toward the ability to reason (whether quantitatively or qualitatively) based on quality attribute-specific models. The goals of having a knowledge base are to make architectural design more routine and more predictable, to have a standard set of attribute-based analysis questions, and to tighten the link between design and analysis.

The reusability of existing knowledge includes packages of analysis templates associated with reusable conceptual architectures. The architecture-specific experience must be structured in a knowledge base so as to provide a set of pre-packages analyses and questions including known solutions to commonly recurring problems and known difficulties in employing those solutions. A PLA analysis knowledge base should be organized in three important sets:

- A set of materials that describe many of the evaluation artifacts,
- A set of quality attribute-specific questions that aid the evaluator in probing a product architecture,
- A set of questions that aid the analyst in gathering the information needed to build an analytic model of the quality attribute.

After this phase, activities of concrete architecture design and the second phase of architecture analysis are performed.

3.4 Concrete architecture design

The concrete software architecture provides the hierarchical containment of concrete software components and definition of communication protocols used between components. 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 three similarly named architecture views as in conceptual architecture: structural view, behavior view and deployment view. In this concrete approach to architecture, every view produces its architectural descriptions (Figure 6).

Architectural styles selected for the conceptual architecture are now complemented by design patterns [23]. These micro architectural elements [15] guide designers of components in greater detail to implement the components and services compatible with the architecture. Appropriate design patterns are recorded as the 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 means of refining components and interfaces between them. The hierarchical structure is illustrated in structure diagrams, which are built up with respect to refined non-functional requirements.

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

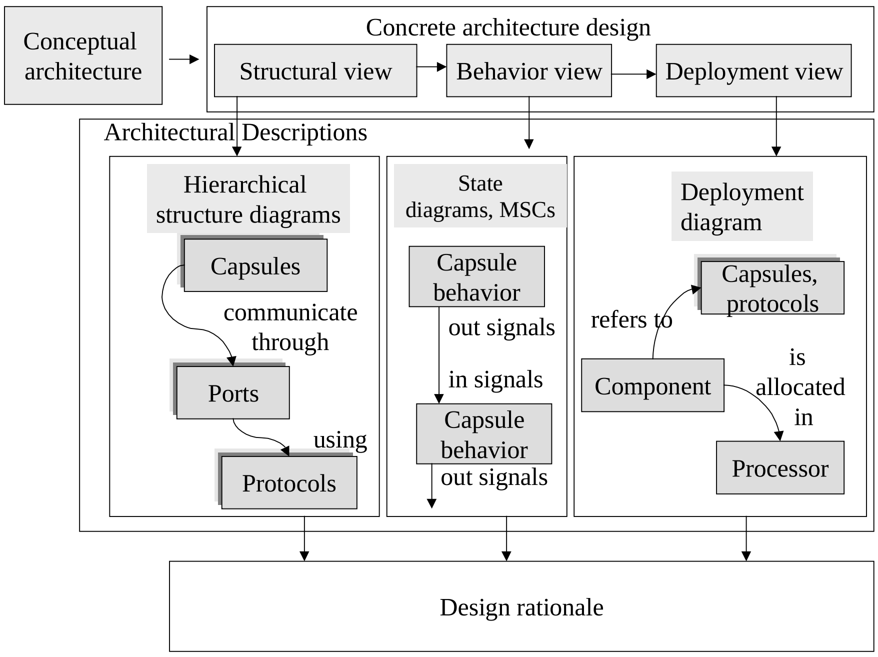


Figure 6. Design phases and architectural descriptions of the concrete architecture.

The third concrete view creates executable software components that refer to concrete architectural components, i.e. to capsules and also to protocols.

Software component instances can be allocated to hardware and illustrate this kind of deployment in a system level diagram.

The modeling elements and diagrams mentioned above mainly follow the Unified Modeling Language specification version 1.3 [55]. Structural modeling elements supporting component-based development and run-time architectures are mandatory requirements for UML version 2.0 [62]. A suggestion for a possible notation for component-based software development is introduced in [57].

3.5 Concrete architecture analysis

Concrete architecture analysis focuses on drivers for architectural development. In this phase recommendations are made, “hot spots” in the architecture (areas of high predicted complexity, large numbers of changes, performance bottlenecks, etc.) are located and strategies for their mitigation are enumerated, common reference models are identified. It is important that this phase ties back to the stakeholders’ values, as they are the drivers of the analysis in the first place. Stakeholders' values measure their commitment to the product line architecture.

Scenario-based assessment is particularly appropriate for qualities related to software development, which are specific to product-line architectures. Software qualities such as maintainability, reusability, modifiability, adaptability and portability can be expressed very naturally through change scenarios. However, scenario-based evaluation depends on the objectivity and creativity of the analyst who defines and executes them. Still, the use of scenarios for evaluating architectures is recommended as one of the best industrial practices.

This phase is based on SAAM [9], but improved through the introduction of a *knowledge base* for product-line architecture design and analysis and *categories of scenarios*. The knowledge base is attached to the product-line scope in form of requirements’ taxonomy. The analysis phase considers product line specific techniques such as commonality analysis, which systematically models the required similarities and differences among product line members. It is also considered that the product-line architecture contains the common components of the architectures of the product members and takes variabilities as possible changes of this. The goal is to establish how adaptable the product-line architecture to the expected changes related to this taxonomy is.

The analysis method consists of five important steps (Figure 7), which are: *deriving of change categories from the product line scope, product-line architecture description, scenarios identification, evaluation of the effect of the scenarios on the architecture elements* and *scenario interaction.*

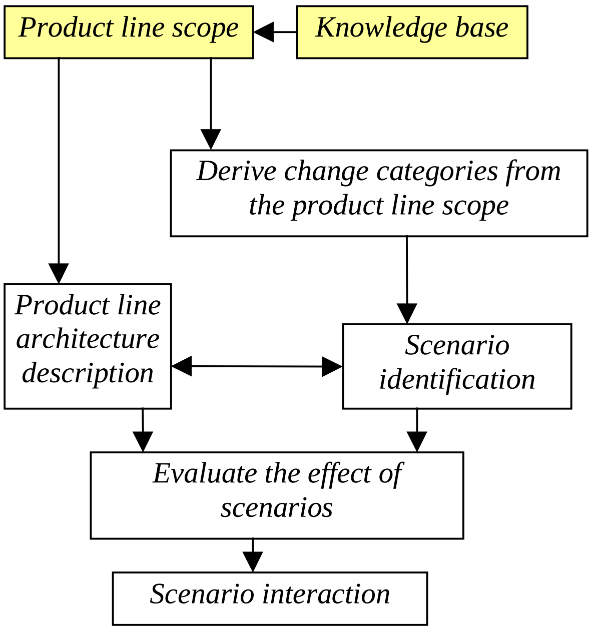

Product line scope
Knowledge base
Derive change categories from the product line scope
Product line architecture description
Scenario identification
Evaluate the effect of scenarios
Scenario interaction

Figure 7. Inputs and activities of architecture quality analysis (AQA).

The main inputs of the method are the product-line scope and product-line architecture. The first step is to derive change categories based on the product-line scope. Then, in parallel, the description of the product-line architecture and scenarios identification are performed. The simultaneity of these steps permits deciding what view should be considered for an elicited scenario. Scenarios effects evaluation and scenario interaction are the last two steps performed sequentially.

4. Requirements engineering

The purpose of requirements engineering here 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. Requirements engineering for a product line architecture is done in steps, which are:

- Requirements specification
- Requirement analysis
- Defining context
- Scoping the product line

These steps of requirements engineering are described next.

4.1 Requirement specification

First, the system is defined in the form of *main requirements*. These requirements are specified on high level for the whole system. Requirements i.e. *technical properties* of the system equal the main *quality* and/or *functional* goals this future system has to provide. Technical properties identified in this early phase of architecture design drive the design process and are described in textual form.

In addition to functional and/or non-functional requirements, *the system constraints* should be defined in this phase. Constraints mean here specific *standards* or *rules set by laws* that constrict the development process of software architecture. In addition to the standards and rules set by laws some quality attributes may also restrict the design of the architecture and should be taken into account in this early phase.

The next step is to analyze what do the technical properties and constraints actually mean, and how do they affect the design of the software architecture.

4.2 Requirement analysis

Requirement analysis step is to analyze the technical requirements, i.e. properties and constraints, defined in the requirement specification step by refining the effect of each requirement on the architecture and determining the possible *prerequisites*.

By prerequisites it is meant that requirements often depend on each other and in order to fulfill one property, the system might need to meet some other requirements.

4.3 Defining context

While capturing the technical properties of the system it is important to *document the context* i.e. the interfaces to the environment. The system, its environment and interfaces between these two are defined in this phase with a context model. The context model represents the system and its environment drawn up with freely formed notation.

System context usually represents a high level view to the issue, but the abstraction level can be specifically defined for each case. It should be kept in mind that the purpose of this phase is to describe the system as a whole in its environment, not to decompose the system into subsystems, yet. System context may include e.g. location of the system in its environment and/or a rough distribution of the system in the case of a distributed environment.

4.4 Scoping the product line

Before starting the development of a product line, a company has to consider various issues in order to get an understanding of whether a product line is appropriate for different technologies and businesses [47]. In order to help make decisions about product-line scope, an evaluation that considers an appropriate value metric is needed. The analysis of architectural qualities may be driven with scenarios that identify which changes are most valuable to a customer and quantifies the expected return on making that change. Since it is developing a common architecture for a family of products, the goal is to design an architecture that encompasses all the product-line members' common features, but which can be easily adapted to produce any member of the family. This means that addressing the variations among members should require no change, or very little change to the common architecture.

An important stakeholder in product-line scope analysis is the customer. A customer value analysis approach is similar to the quality function deployment method (QFD) [25] in seeking to reconcile customer needs with product design. It differs in that, it seeks to measure more directly and accurately the customer's perception of total delivered value, hence what the customer will actually pay for the product. Second, it directly measures the difference between an organization's internal perception of customer value and the customer's actual perception. This provides a basis for aligning a company internal view of delivered value with market realities.

The idea is to provide a common framework and metric for making decisions those bring together business issues and product issue. To do so a value metric that makes sense in all these contexts is required. A value metric is applied to help make decisions about product-line scope. The foci are on:

- Measuring the relative benefit of including or excluding capabilities from a product-line's scope.
- Making cost/benefit decisions about adopting product-line solutions.
- Applying product-line technologies (e.g. building a compiler for a product line).
- Developing a particular product-line.

The purpose of the PL scope evaluation is both to assess the quality of a given architectural design relative to the design goals and to quantify that measure of quality so it can meaningfully compare different designs. This could be done using the results of domain suitability (see Figure 8).

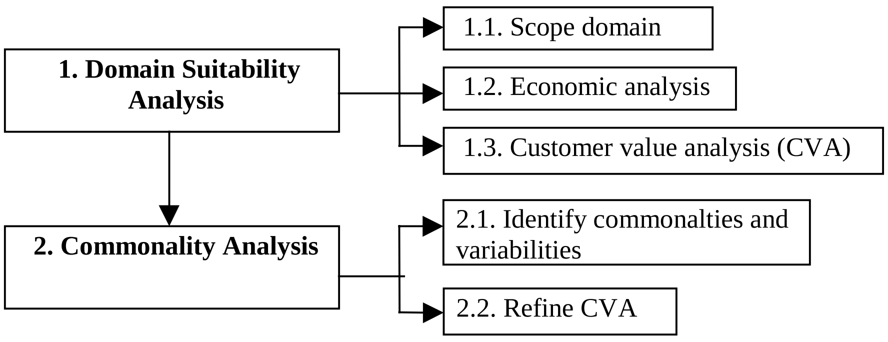

1. Domain Suitability Analysis
1.1. Scope domain
1.2. Economic analysis
1.3. Customer value analysis (CVA)
2. Commonality Analysis
2.1. Identify commonalties and variabilities
2.2. Refine CVA

Figure 8. Using customer value in defining the scope of product line.

Domain suitability. The activities are domain scope, economic analysis and customer value analysis (CVA).

- Domain scope considers the creating of a preliminary definition of the product-line in terms of commonalties and variabilities.

- Economic analysis is concerned with the building of an economic model of the product's cost/return using the company's current software products and their product-line; these models may be used to determine the expected return from adopting a product-line approach.

- Customer value analysis (CVA) is performed to help establish the relative value of the possible variations in the potential scope of the product line. In this context, change scenarios can be constructed based on how the product is expected to evolve to meet customer needs. CVA results are used both to identify which changes are most important (valuable) and to quantify the expected return on making the change.

Commonality analysis. This activity has the goal to identify and document the commonalties and variabilities characterizing the software product-line. Also, the CVA is refined by developing value metrics based on the more detailed definition of expected variations. This aligns the customer value data with the product-line requirements that will drive a domain engineering phase.

5. Conceptual architecture design

Conceptual software architecture provides the organization of functionality and quality responsibilities to conceptual structural components, collaboration between components and allocation of components to hardware.

These different aspects of conceptual software architecture are represented with three architecture views: structural, behavior and deployment view, which are next described in detail.

5.1 Conceptual structural view

Conceptual structural view is next described by means of architectural elements, language and design steps. *Architectural elements* are created and used in architectural descriptions of structural view. *Language* provides something concrete to describe the elements represented and *design steps* refer to activities that are carried out during the development of conceptual structural architecture.

5.1.1 Architectural elements

The conceptual structural view is used to record the architectural elements: *conceptual structural components*, *conceptual structural relationships* between components and *responsibilities* these elements have in the system (Table 2).

Conceptual structural components divide the system into functional blocks. These elements can either be classified with a <<system>>, <<subsystem>> or a <<leaf>> stereotype label. At the top level, the system is decomposed into subsystems and at the next level, the conceptual subsystems are decomposed into conceptual leaf components. At the end of the decomposing process, conceptual leaf components are the smallest blocks used on the conceptual architecture level.

The conceptual components of the higher level form a clustered set of functional properties and are therefore called subsystems. In order to reach high quality software architecture these components on the subsystem level should have a low coupling among each other.

Table 2. Conceptual structural design elements, types and responsibilities.

Element	**Types of an element**	**Responsibilities of an element**
Conceptual structural component	System component	What it has to do?
	Subsystem component	
	Leaf component	How it does it?
Conceptual structural relationship	Passes-data-to	
	Passes-control-to	Why? (Design Rationale)
	Uses	

Coupling is an ordinal scale describing the degree of interdependence between architectural components. Low coupling means that if one component of a program is changed, it requires changes to be made to as few other components as possible in the system [59].

Conceptual leaf components on the lower hierarchical level perform smaller groups of functional requirements inside the subsystem. The components inside one subsystem, instead of two subsystems, should have a high coupling with each other.

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.

Passing *control* means, that one component controls a given aspect of the system. The control component receives its inputs from the external environment and generates outputs to the external environment, without any human intervention [24]. By an external component here is meant the environment of the component i.e. other components or subsystems. A control component is often state-dependent. In some cases, some input data might be gathered by some other component(s) and used by this component. Alternatively, this component might provide some data for use by other subsystems.

Passing data means, that one component inputs data to another component, but is still able to continue processing without waiting for a reply or a return value. The situation is opposite when talking about *uses* relationships, where a component has a kind of subcontract with the component it uses. The user component asks another component to process some data for it and is forced to wait for the return value.

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

5.1.2 Conceptual structural language

This section provides a suggestion for a conceptual structural language. That is, a concrete means of describing the elements represented earlier i.e. *conceptual structural components* and *relationships*. Figure 9 shows how conceptual components can be described using graphical language.

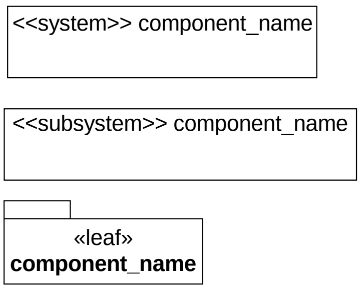


Figure 9. A description for conceptual components.

The aggregation level of each component is highlighted with a stereotype label. Containment is represented by placing components on the lower aggregation level inside the components on the higher aggregation level. That is, for instance, a package symbol inside the rectangle that represents a subsystem.

Conceptual structural relationships can be described as shown in Figure 10. Different types of relationships are identified with stereotypes.

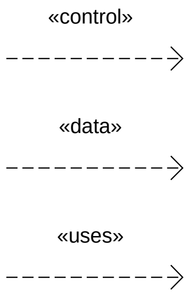


Figure 10. A description for conceptual relationships.

A relationship arrow can connect components placed on different or on the same aggregation level. This depends on system size and complexity. For instance, in small and not so complex systems, a relationship arrow can appear between two leaf components, meanwhile describing large systems, two subsystem components are connected with a relationship arrow.

More complex situations are easier to figure out, if a leaf component is connected to the subsystem representing that the leaf component has a relationship (e.g. passes data to) with every leaf component in that subsystem (Figure 11).

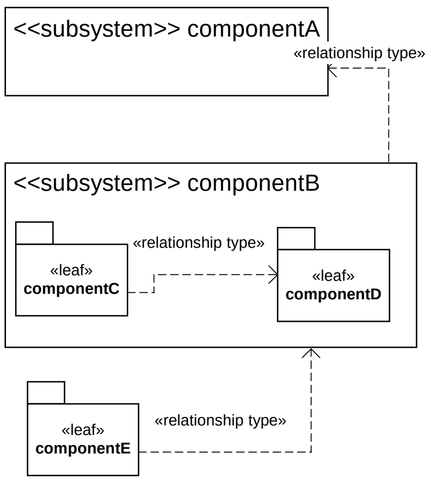


Figure 11. Usage of conceptual structural relationship.

5.1.3 Design steps of conceptual structural view

The design of the conceptual structural view is iterative and can be described with a triangle including three steps, which are done with respect to constraints set in the requirements engineering, as shown in Figure 12. Thick arrows represent the inputs for design steps and the block arrows represent interaction between the design steps.

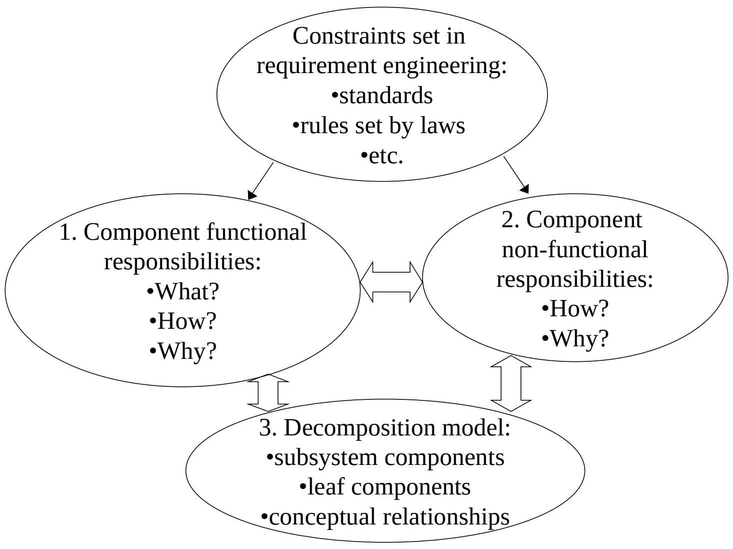

Constraints set in requirement engineering:
•standards
•rules set by laws
•etc.
1. Component functional responsibilities:
•What?
•How?
•Why?
2. Component non-functional responsibilities:
•How?
•Why?
3. Decomposition model:
•subsystem components
•leaf components
•conceptual relationships

Figure 12. Design steps of the conceptual structural view.

The three design steps that result in the conceptual structural architecture are discussed next.

Step 1: Define functional responsibilities

In this, the system functionality is captured as a textual responsibility list. Because requirements may not be fully known independently of a solution, in the beginning the textual list of system responsibilities is a sketch of planned functionality. This list is processed forward and requirements may change frequently, especially when a new system is under development.

The purpose of a responsibility list is to capture system functionality at an abstract level and encapsulate functional properties into groups. It is unlikely that designing a structural view does not require iterations to optimize the composition. During the iterations (done always through steps 2 and 3, defined later in this report) the list gets more shape. Use of text processing effects helps to outline the list and brings out the sub-lists.

The technical properties of a system, defined in the requirements engineering phase, have also to be considered here. The results of the requirements engineering phase are taken into account when defining the functional responsibilities of a system. Functional responsibilities answer the questions ‘what’, ‘how’ and ‘why’. The first two questions come to an end in a structural model with selected architectural styles, while ‘why’ gets its form as recorded design principles and rules, i.e. design rationale.

Step 2: Define quality responsibilities

Quality responsibilities are derived from the non-functional requirements by answering the question ‘how fast’, ‘how often’, etc. Constraints set in the requirements engineering phase have impact an on defining these responsi-bilities. Checking the balance between functional and quality responsibilities is essential in order to find out conflicting responsibilities.

Non-functional responsibilities are listed in a table and categorized according to quality attributes. A single quality attribute may have several *non-functional responsibilities* included.

This phase tries to encapsulate the quality requirements as a responsibility of one conceptual component or at least some restricted part of the architecture. This leads to applying a specific architectural style (selected in Step 3) with respect to a desired quality attribute in that component.

Step 3: Cluster responsibilities to components and subsystems

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

The final third step in the structural design is to *group responsibilities* to conceptual components. In addition, the quality attributes drive the *selection of architectural styles* and thus affect the forming of the system architecture.

Different architecture styles [9], architectural patterns [10] or mechanisms, are addressed to certain selected quality attributes. In practice, these qualities are not exclusive. A software product has to meet the requirements of various kinds of quality attributes at the same time and this leads unavoidably to the use of heterogeneous architectural styles. Heterogeneity means that different styles are used for parts of the system in a beneficial way.

Table 3 provides some of the commonly used architectural patterns and what quality attributes they help to achieve [10].

Table 3. Sets of architectural patterns.

Performance	**Modifiability**	**Security**
Load balancing	Data router	Encryption
Priority assignment	Data repository	Integrity
Fixed priority scheduling	Virtual machine	Firewalls
Cyclic Executive scheduling	Interpreter	Mirroring of databases
Client-Server		Audit trail
Availability	**Testability**	**Usability**
Ping/Echo	Monitors	Separation of command from data
Voting	Backdoor	Separation of data from the view of that data
Recovery blocks	Open APIs	Replication of commands
Atomic transactions		Recording
Checkpoints		Explicit models for Task, User, System

5.2 Conceptual behavior view

The conceptual behavior view is next described by means of architectural elements, language and design steps. *Architectural elements* are created and used in architectural descriptions of behavior view. The *language* provides something concrete to describe the elements represented and *design steps* refer to activities that are carried out during the development of conceptual behavior.

5.2.1 Architectural elements

The conceptual behavior view is used to specify the behavior of a system on a high abstraction level. Behavior is recorded by defining *conceptual behavior elements: components* and *relationships* and the responsibilities these elements have in the system (Table 4).

Table 4. Conceptual behavior design elements.

Element	Types of an element	Responsibilities of an element
Conceptual behavior component	Service component	What it has to do? How it does it? Why? (Design Rationale)
Conceptual behavior relationship	Ordered sequence of actions	

Conceptual behavior 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 behavior relationships between each other. Behavior 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.

5.2.2 Conceptual behavior language

This sub section introduces a language for the conceptual behavior of an architecture, i.e. a concrete means of describing the entities represented earlier i.e. service components and ordered sequences of actions. Figure 13 illustrates a graphical representation of conceptual behavior components i.e. services.

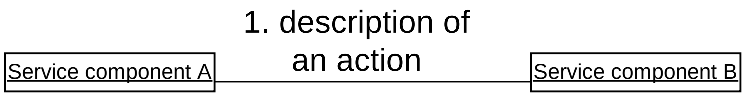


Figure 13. A description for conceptual behavior elements and ordered sequence of actions.

Ordered sequences of actions are represented with textual descriptions of actions that are suitably numbered. Direction of an action is shown with an arrow. To describe collaboration scenarios that are related to the user interface, an actor called e.g. “user application” may be useful.

5.2.3 Design steps of conceptual behavior view

The design of the behavior view can be described with three steps, which are started with functional responsibilities as input, as shown in Figure 14.

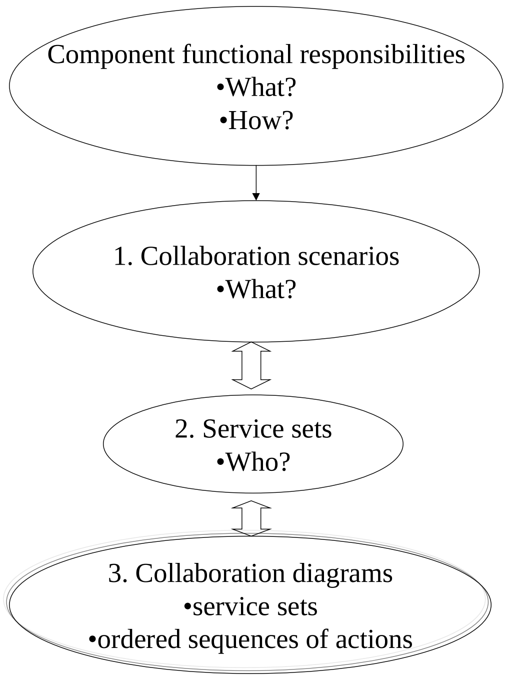

Component functional responsibilities
•What?
•How?
1. Collaboration scenarios
•What?
2. Service sets
•Who?
3. Collaboration diagrams
•service sets
•ordered sequences of actions

Figure 14. Design steps of the conceptual behavior view.

The three design steps that result in a collaboration model are discussed below.

Step 1: Select collaboration scenarios

Here, the system abstract behavior is ascertained by deriving special collaboration scenarios from the functional requirements of the system. These scenarios are the most essential sequences of actions in the system.

The purpose of collaboration scenarios is to be an intermediate phase in the design, quid the architects in the design process of the behavior view and finally produce collaboration diagrams (Step 3). Collaboration scenarios are documented in the form of a table, but they are not included in the final architectural documentation i.e. architectural descriptions, as collaboration diagrams are.

Step 2: Identify service sets

In this step, special *service sets* related to collaboration scenarios are identified by answering the question 'which services are participants in this scenario'. Service sets are collected in a table with collaboration scenario descriptions.

Step 3: Create collaboration model

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.

5.3 Conceptual deployment view

The conceptual deployment view is described next by means of architectural elements, language and design steps. *Architectural elements* are created and used in architectural descriptions of deployment view. The *language* provides something concrete to describe the elements represented and *design steps* refer to activities that are carried out during the development of the conceptual deployment view.

5.3.1 Architectural elements

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 5).

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 5. Conceptual deployment design elements.

Element	**Types of an element**	**Responsibilities of an element**
Deployment node	(Various, depends on system)	What it has to do? How it does it? Why? (Design Rationale)
Unit of deployment	Mandatory Alternative Optional	
Conceptual deployment relationship	Is-allocated-to	

Each unit of deployment represents one of three alternative types. These types are *mandatory*, *alternative* and *optional* [35, 46]. *Mandatory* units are the basic building blocks of the software. These units are always included in a system. *Alternative* units cannot concurrently exist in the system i.e. there is a mutual exclusion relationship between these units. *Optional* type means that the unit is not essential for the system functionality, but may be included in order to reach optional functionality and variability.

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

5.3.2 Conceptual deployment language

This section defines how to describe deployment nodes, different types of units of deployment and allocation relationships in a conceptual deployment view. *Deployment units* are rectangles that are differentiated with fills and line patterns. A hardware *node* is a rectangle with heavy line weight and connections between nodes can be represented with two-headed arrows. Placing deployment unit rectangles inside the node rectangles represents allocation relationships.

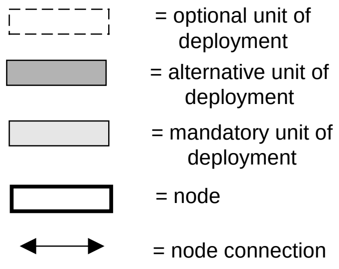

= optional unit of deployment
= alternative unit of deployment
= mandatory unit of deployment
= node
= node connection

Figure 15. A description for conceptual deployment view.

5.3.3 Design steps of conceptual deployment view

The design of the deployment view can be described with three steps, which are started with leaf components as input, as shown in Figure 16.

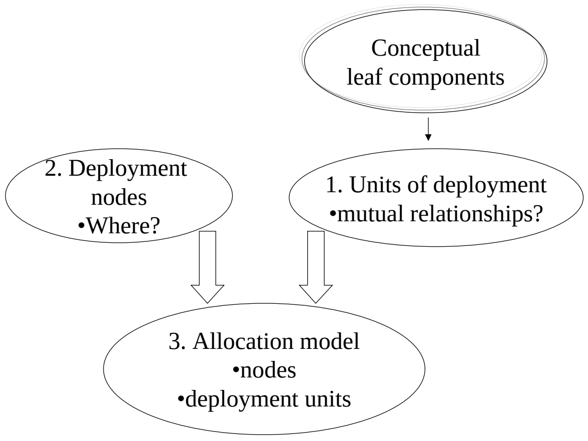

Conceptual leaf components
2. Deployment nodes •Where?
1. Units of deployment •mutual relationships?
3. Allocation model •nodes •deployment units

Figure 16. Design steps of conceptual deployment view.

The three design steps that result in an allocation model are as follows.

Step 1: Cluster leaf components to units of deployment

The aim of the first design step is to cluster leaf components into atomic deployment units. Clustering is done mainly by identifying mutual relationships between leaf components but a single leaf component may also compose a deployment unit. Leaf components that require each other in order to function properly build up a deployment unit. A deployment unit has to be small enough to reach atomicity in the allocation process.

The types of deployment units are defined either according to system variability or with respect to leaf component relationships. For example, mandatory and optional types are often defined based on variability. *Mandatory* units of deployment form the basis for services in one node and *optional* deployment units represent additional services.

In addition to mandatory and optional deployment units there may exist deployment units of an *alternative* type. The alternative type often represents a mutual exclusion relationship between two or more leaf components in separate units of deployment (Figure 17).

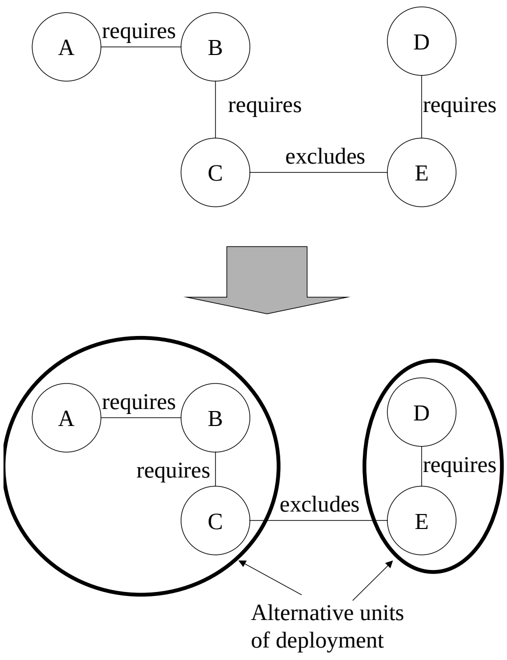


Figure 17. Clustering leaf components according to mutual relationships.

Step 2: Identify deployment nodes

In this step the deployment nodes are identified. A definition for a node in a system is considered. A definition might be, for instance, a device with multiple processors, a device with single processor or just a processor in an undefined gadget. Is there a distributed environment in question and what are the associations between nodes? The multiplicity of nodes has also to be taken into account. Node definition is equal to the type of node.

Step3: Allocate deployment units to nodes

In the following, deployment units are allocated to nodes and represented in the form of an allocation model. Figure 18 illustrates the concept of allocation. An allocation model describes the types of deployment units graphically and provides a coarse- grained sketch of the system hardware.

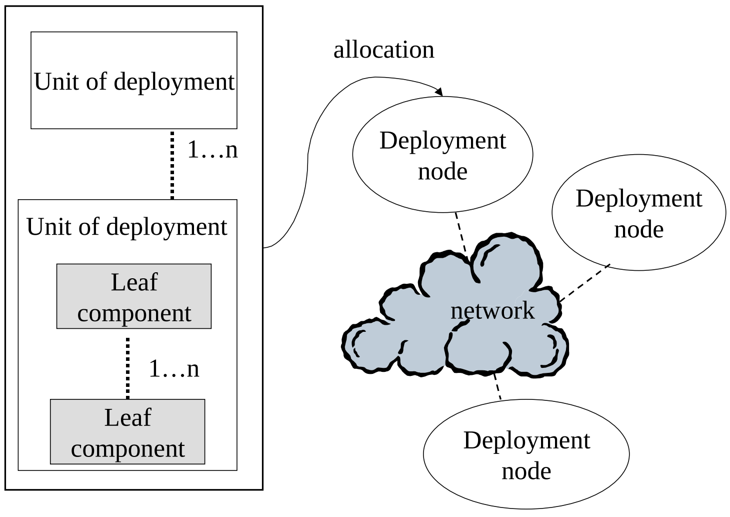


Figure 18. Allocation of deployment units into nodes.

6. Quality analysis of the conceptual architecture

In quality analysis of the conceptual architecture, commonalty analysis (see Figure 8) started in the scoping of the product line is continued by focusing on the way variability and quality are addressed in the conceptual architecture. Among the specific points that deserve special attention are the kinds of variation which can be covered by the architecture and properties that are preserved for all variants of an architecture, stability of component interfaces with respect to evolution in products. Variation points and product-line patterns are the main interests in the variability analysis that is continued by analysing the architectural styles and patterns selected for the product-line architecture.

6.1 Variability analysis and representation

In the context of software product line modeling variability is essential for building a flexible architecture. It is possible to anticipate some common and variable aspects in the requirements of different product members and to construct a product member in such a way that it facilitates this type of variability and it considers strategic reuse. A product-line architecture is based on a family of similar products, so it includes the complete specification of those parts that are common to all products. However the points where products differ from each other must be represented, in order to prepare the product-line architecture for configuration. Showing variability explicitly concentrates the places where product specific adaptations have to be made, and ensures that structures that need not be changed are preserved. It should be noted that, when talking about variability in respect to product-line architectures, one must distinguish between variability in the domain that is resolved at a derivation time and variability in single derived product architectures that are evaluated at runtime. In this context the focus is on the representation of domain (i.e. derivation time) variability, as solutions for modeling runtime variability are already available in the form of design patterns.

It is still a challenge to create malleable software architecture [53], even if the problem is quite old [48]. Representation of variability inside a PLA has been done in the following ways:

1. Variability appears implicitly [44];

2. Variability is abstracted in the object-oriented class structure [34];

3. Variability is performed during domain engineering using Scope, Variability and Commonality (CSV) analysis [16];

4. Variability is represented by using a hot spots in frameworks [50, 22];

5. Variability is represented using patterns [42, 39];

6. Variability is represented by the means of under-specification (leaving everything that varies unspecified), provisions (letting one select among possible variants) and patterns (structuring the architecture for easy adaptation).

Variability is modeled on the domain architecture level and is finally resolved by derivation of a specific product. Variation in product-line architecture could be managed in the two ways identified in [17] as following: variation point description and product-line patterns. In treating the variability issue it is important to clearly separate three main concerns:

1. Expression of *what* can vary in the product-line architecture (the implied assets) and *how* it can vary (description of all kind of variants).

2. Explicit signalization of places in the model (hotspot) *where* variation can occur (variation point). Each variation point is considered as an elementary entity (it is not decomposable in sub entities) and is self-contained (all needed information to choose a variant locally is provided). If this point is not identified, teams will have to adapt or alter the assets in ways that may not have been anticipated, leading to product line erosion.

3. The decision to take *when* facing variants selection (the decision model).

Variation points are exhibited using a hot spot in a UML class or package diagram. It is relatively easy to describe the variability point in UML models. The major difficulty consists in describing the decision model.

A solution for *product-line patterns* is to integrate under-specification into variability. This means that abstract components are leaf nodes that define the interfaces for product-specific additions. Consequently, abstract components represent points in the product-line architecture that must be further specialized by configuration or extended by product-specific parts, whereas concrete components denote commonalties in the domain. The meaning of concrete and abstract in this context should not be confused with the meaning of concrete and abstract in object-oriented programming. In this pattern context, “concrete” components just mean parts that are completely specified and “abstract” components mean parts where things have been left open. Nevertheless, abstract components can be instantiated, but must be further specified if they are to be implemented.

6.2 Architecture analysis at the conceptual level

This section introduces quality attribute-based methodology and architectural styles and patterns violations analysis.

6.2.1 Quality attribute-based methodology

In addition to variability analysis, the activity in this step is based on the structured relation between the software quality attributes of a system and the system architecture. In order to apply an unifying approach for reasoning the multiple software we assume that the properties of critical systems and the best methods to develop them already exist and concern different schools/opinions/traditions (e.g. performance – from the tradition of hard real-time systems and capacity planning).

Quality analysis method at the conceptual level is based on a generic taxonomy of attributes and connections between the attributes [7]. At the conceptual level, the quality attribute taxonomy identifies:

- *concerns* – the quality requirements by which the attributes of the PLA are judged, specified and measured (i.e. throughput for performance, observation interval, etc);

- *attribute-specific factor* – properties of the architectural styles and patterns and environment that have an impact on the concerns (i.e. separation, replication, resource sharing, faults);

- *methods* – how the concerns are addressed; analysis and synthesis processes during the development of the system (e.g. the QADA method, formal methods, fault tolerance methods), users' guide and training.

The investigation of the relationships between quality attributes and software architectures leads to an attribute-based methodology for evaluating software architecture.

The structural view, the behavioral view and deployment view are the descriptions the analysis is based on. A mapping between the functional requirements and subsystems or components provides a *structural* view that supports *traceability* analysis, especially if there is any *modification* to be made to the system. Two different representations may be considered for the mapping. One shows the mapping of the system’s main functional and non-functional requirements or features to subsystems and components, whereas the other demonstrates the mapping of components to features. The conceptual components involved are linked back to those shown in a structural view. The first representation helps locate all components involved for a particular functional requirement. The book-keeping effort in creating and maintaining such view, and the links between them, is crucial for supporting analysis. Humans can not be expected to keep all the details in their heads, all the time.

A *behavioral view* represented by collaboration diagrams is perceived as a dynamic aspect that it is important to understand the system before a reuse occurrence. In addition, the behavior view also supports *maintainability* as a system evolves. For instance, if modifications are made, the structural architectural representations may stay the same, but some of the system’s behaviors may be modified. The *modification* of behaviors should be, but typically can not be, explicitly represented by structural architectural views.

Another example is that if personnel changes or the architect leaves, there may be different interpretations for the structural view by other designers or new employees.

The *deployment view* is also incorporated for this conceptual analysis although the main objective focuses on the reuse and adaptability perspective. The deployment view is the description of allocation, and sets constraints that have an impact on quality requirements, i.e. concerns.

Based on the architecture descriptions, during this phase we use the taxonomy of quality requirements to *identify architectural styles and pattern violations*.

6.2.2 Architectural styles and patterns violations analysis

An architecture can be classified into more than one style and an architecture allows coexistence of multiple styles. The primary purposes of the style or pattern is to impose an overall structural interpretation on a software system or subsystem for consistency checking, and to support the human to human communications of the software. The identification of an architectural style helps focus on critical features such as the control mechanism of a style, the communication mechanism between components, and the integrability of new components, or the modifiability of existing components.

In addition, the analysis can support a decision-making process in choosing an appropriate style for the product-line or trade-off analysis. The appropriate style can then be reused for the product-line domain, even if the architecture itself is evaluated to be risky to be directly reused for a product. For large systems where multiple styles may exist, analysis of styles interoperability is important. Styles interoperability is directly related to system integrity and maintainability. It is important to identify and analyze how one particular style communicates with other styles.

This activity deals with the components or links that are missing or are not represented properly, and the control or communication mechanisms that violate the policy of the identified architectural style. The architectural style may only reveal an “idealized” or “as-intended” software architecture initially developed by a group of software designers. This activity, on the other hand, recovers the “as-built” aspect of an architecture supported by the behavior representations.

Some reasons for the violations could be legacy systems, modifications for performance, understandability, and discrepancies in the levels of abstraction. The violations must be explicitly documented to reduce potential problems caused by ambiguity or inconsistency. The documentation can also support system maintainability. Architectural violations are as important as normal architectural features and must be identified before reuse occurrence to reduce unnecessary maintenance effort.

7. 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, behavior view and deployment view. These views are next described in detail.

7.1 Concrete structural view

Concrete structural view is discussed below. It is described by means of architectural elements, language and design steps. *Architectural elements* are created and used in architectural descriptions of the structural view. *Language* is a way to describe the elements represented and *design steps* refer to activities that are carried out during the development of concrete structural architecture.

7.1.1 Architectural elements

The concrete structural view is used to record concrete architectural elements: *capsules, ports* and *protocols* (Table 6).

Capsules are the fundamental modeling elements of real-time systems [52, 57, 51]. A capsule represents independent flows of control in a system. Messages are the sole means of communication between capsules and they are sent and received through ports.

Capsules represent the system with hierarchical aggregations of capsules. The upper level capsule contains lower level capsules and is therefore often called *a container* capsule. The uppermost container capsule is called *a top capsule.* Capsules that compose a top capsule, are labeled with <<subsystem>> stereotype to represent *subsystem capsules* and capsules on the next *aggregation* level are *component1 capsules*, *component2 capsules* on the next level etc., although three composition levels are deep enough in most cases.

Table 6. Concrete architecture design elements.

Element	**Type of an element**	**Roles of an element type**	**Visibility**	**Responsibilities of an element**
Capsule	Top capsule Subsystem capsule component1 capsule component2 capsule	Fixed Optional Plug-in	–	What? Why?
Port	Wired Non-wired	In Out	Public Protected	What protocol is used?
Connector	–	–	–	Which ports does it connect?
Protocol	Binary protocol	Base Conjugate	–	What messages are exchanged in this protocol?

Capsule roles represent a specification of the type of capsules that can occupy a particular position in a capsule's structure. Capsule roles are strongly owned by the container capsule, and cannot exist independently of the container capsule. A capsule's structural decomposition usually includes a network of collaborating capsule roles joined by connectors. Capsule roles are classified into three categories: *fixed*, *plug-in* and *optional* capsule roles. Capsule roles reflect component types in conceptual architecture.

Fixed – By default, capsule roles are fixed, meaning that they are created automatically when their containing capsule is created, and are destroyed when the container is destroyed.

Plug-in – The structure of a capsule may contain plug-in capsule roles. These are, in effect, placeholders for capsule roles that are filled in dynamically. Plug-in roles can be defined as substitutable components in architecture.

Optional – Some capsule roles in the structure may not be created at the same time as their containing capsule. Instead, they may be created subsequently, when and if necessary, by the state machine of the capsule. Optional capsule roles can be destroyed before the container is destroyed and these roles can be defined as substitutable components in the architecture.

Capsules do not support visibility and their responsibilities are captured similarly as for conceptual structural components by answering the questions 'what', 'why' and 'how'. 'Why' denotes the *design rationale* and 'how' is mapped to *variability* i.e. to the role capsule represents.

Ports are objects whose purpose is to send and receive messages to and from capsule instances. They are owned by the capsule instance in the sense that they are created along with the capsule and destroyed when the capsule is destroyed.

Wired ports must be connected with a connector to other ports in order to send messages. *Non-wired* ports are used to model dynamic communication channels. These ports do not have to be connected with *connectors* to other ports. *A connector* is a concrete structural relationship between capsules.

Port role represents the direction of messages exchanged through a port. A port is a *participant* in a communication relationship and can play two alternative roles, *in* or *out*. Both types of signals are always exchanged, in-signals and out-signals, through a port and this does not depend on the *role* port is playing. The first message delivered through a port determines the role of a port.

Ports can also be classified according to the *visibility* of their support. *Public ports* are ports that are part of a capsules interface and may be visible both from outside the capsule and inside. *Protected ports* are used to connect capsules to contained capsule roles. These ports are not visible from the outside of a capsule, since they are not part of the capsule's interface.

The *responsibility* of a port is to deliver messages of a certain defined *protocol*. An *in* port plays a *base* protocol role and correspondingly an *out* port plays a *conjugate* role in a *binary protocol*.

A protocol is a contractual agreement defining the valid types of messages that can be exchanged between the participants in the protocol. *A binary protocol* is a set of messages exchanged between *two* participants. A binary protocol needs only a *base* role to define all the messages that are exchanged in a protocol. The other role, called the *conjugate*, can be derived from the base role simply by inverting the incoming and outgoing message sets. This inversion operation is known as *conjugation*.

7.1.2 Concrete structural language

The language represented next is a possible suggestion for a concrete structural language. The concrete structural language, based on the ROOM method [56], described below is supported by Rose-RT [51]. It provides a concrete means of describing the elements represented earlier.

A capsule structure is shown as a box with a heavy border, which represents the capsule's boundary. Capsule types representing the aggregation level of a capsule are illustrated with *stereotypes* in front of the capsule name (Figure 19).

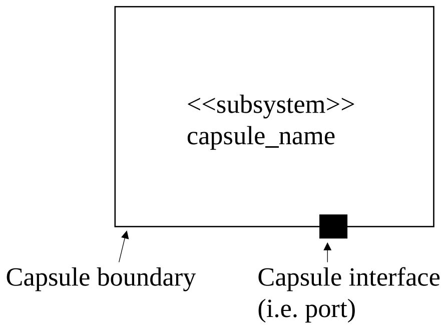


Figure 19. A capsule.

Capsule roles are shown inside the boundary as composite parts. The graphical notation of capsule roles is shown in Figure 20.

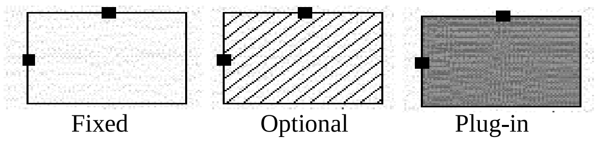


Figure 20. Capsule role classification.

Figure 21 shows a language for describing ports.

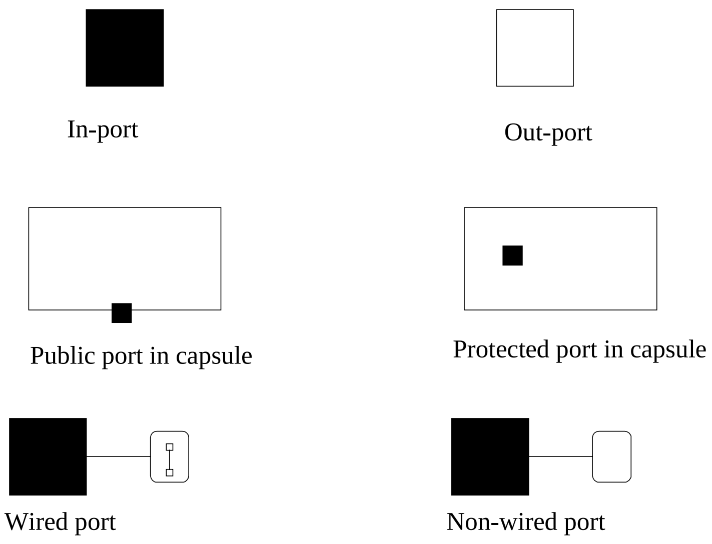


Figure 21. A language for port elements.

Every port has a name and it is shown in a concrete structure diagram. In addition to port name some significant information is related to the port and it is also shown in the structure diagram. Firstly, the visibility of the port is assigned with + and # symbols (+ for public and # for protected port). After the visibility symbol is a place for *the name of the port* and then *the name of the protocol* in use.

Connectors are described as solid lines connecting ports in structure diagrams as shown in Figure 22.

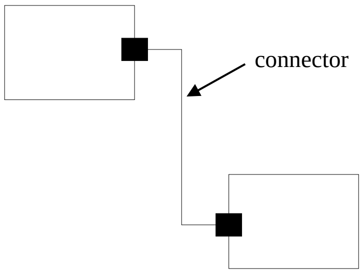


Figure 22. A connector between two capsule ports.

7.1.3 Concrete structural design steps

Concrete structural design conforms to a model of three *parallel* design steps as shown in Figure 23. Concrete design refines the decomposition model built in conceptual architecture design by decomposing it to the lower aggregation levels.

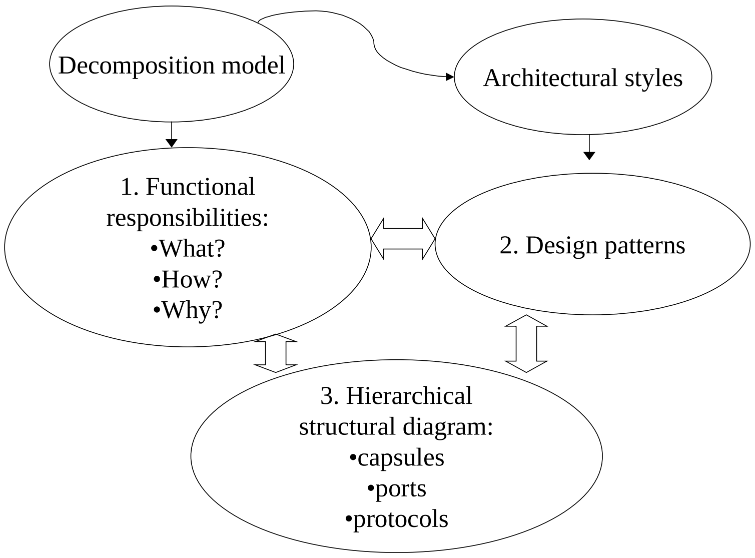


Figure 23. Design steps of the concrete structural view.

Architectural styles selected in the conceptual design phase are input for selecting design patterns as a means of fulfilling non-functional responsibilities. Finally, design patterns and more concrete functional responsibilities take the form of a concrete, hierarchical structure diagram with capsules and ports. These steps are next described individually.

Step 1: Refine functional responsibilities

In this section, the system functionality is refined. The functional responsibilities of each leaf component are expanded in order to create smaller, contained architectural components. Because this step is done parallel with steps two and three, refined functional responsibilities are documented directly to the structural diagram in form of capsule documentation.

Step 2: Select design patterns

Architectural styles selected for the conceptual architecture are now complemented by design patterns [23] that utilize several styles. There is no explicit mapping between styles and patterns and this is why selection of design patterns has to be based mostly on the practical experience of an architect.

Step 3: Build a hierarchical structural diagram

The concrete structural diagram *refines* the architectural structure defined in the conceptual structural view. That is, the concrete top-level structural diagram mainly follows the same structure as defined in the conceptual design. Table 7 shows the comparison of architectural elements used in the conceptual and concrete structural views. As seen, a concrete structural view starts the refinement of conceptual elements at the leaf component level and continues decomposing on as many levels as necessary.

The main refinement from a conceptual decomposition model to a concrete top-level structural diagram is that protocols are defined. In the decomposition model there exist only conceptual relationships, but in the concrete diagram the protocols, i.e. exact signals exchanged through ports are defined as a refinement of relationships. For a signal there are defined its direction, parameters and return values.

Table 7. Comparing conceptual and concrete structural elements.

Conceptual structural view		**Concrete structural view**	
Element	**Description**	**Element**	**Description**
System component	Square		
Subsystem component	Square		
Leaf component	UML package	Subsystem capsule	<<subsystem>> capsule
		level 1 capsule	<<component1>> capsule
		level 2 capsule	<<component2>> capsule
Conceptual relationship	stereotyped UML dependency arrow	concrete interface	ports, connectors, protocols

Capsules themselves remain only as dummy structural components. The composition of each subsystem level capsule is defined in similar structural diagrams at lower architectural levels in the hierarchy.

7.2 Concrete behavior view

The design of the concrete behavior view is next described in three sections. Firstly, the *architectural elements* used in design are introduced. Secondly, a suggestion for a modeling *language* of these elements is described and thirdly, how architectural elements are utilized to build up behavior diagrams. The behavior diagram is build up with the following ordered *design steps*.

7.2.1 Architectural elements

The concrete behavior view is used to model the behavior of a system on a quite detailed level of abstraction. Concrete behavior design elements i.e. components and their relationships that are used in concrete behavior design are defined in Table 8.

Table 8. Concrete behavior design elements.

Element	**Name**	**Type of element**	**Responsibilities of an element**
Concrete behavior component	Interaction instance of a capsule	See Table 6	How does this capsule behave, when a message arrives?
Concrete behavior relationship	Interaction message	Asynchronous message Synchronous message Call message	What protocol signal corresponds to this message? From which port? To what port?

Interaction instance of a capsule conforms to a run-time instance of a capsule representing a certain service. Capsule instance types and roles are presented in Table 6, but those types and roles do not have a great impact when describing behavior inside a capsule or between a set of capsule instances. Behavior inside a capsule instance conforms to the responsibilities of a capsule instance and is represented in the form of a state diagram.

An interaction is a pattern of communication among objects at run-time. A sequence diagram is used to show this interaction from the perspective of showing the explicit ordering *messages*. Sequence diagrams are used to show specific communication scenarios of collaboration.

A message is the specification of a communication between capsule instances that convey information with the expectation that activity will occur upon receipt.

There are up to three types of messages exchanged between capsule instances. *Synchronous send messages*, *asynchronous send messages* and *call messages*. Send messages can be mapped to the *control* and *data* relationships in the conceptual behavior view and call messages can be mapped to the conceptual *uses* relationships.

Synchronous send messages are sent from an instance to another. These types of messages block the sender from waiting for a return (like a function call). This corresponds to the ‘invoke’ operation.

Asynchronous send messages differ from synchronous send messages in that they do not block the sender from continuing its process.

Call message is used between two instances. Call messages are like function calls, so the sender is blocked from waiting for a return.

7.2.2 Concrete behavior language

This section describes how concrete behavior is described using the UML. In addition to the concrete structural language, Rose-RT [51] supports the concrete behavior language discussed below.

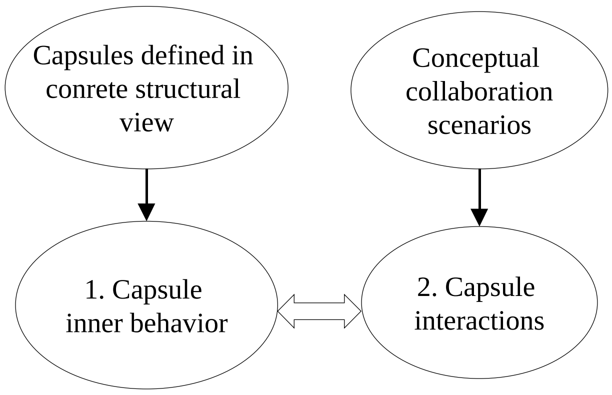
The behavior of a capsule instance is described with a state diagram. *A state diagram* is a directed graph of states connected by transitions. A state diagram describes the life history of instances of a given capsule. A state machine contains exactly one initial state and initial transition, one top state, one or more states, choice points, and the state transitions between them (see Figure 42).

A sequence diagram is used to show the interaction between capsule instances in the form of explicit ordered *messages*. Similarly to collaboration diagrams in the conceptual behavior view, the sequence diagrams in the concrete behavior view are used to show specific communication scenarios of collaboration.

A message instance is shown as a line from the lifeline of one object to the lifeline of another. In the case of a message sent by an object to itself, the arrow may start and finish on the same lifeline. The arrow is named with the name of the message. The arrowhead of the message can be shown in different ways to convey the different types of message communication (see Figure 43).

7.2.3 Design steps of concrete behavior view

The design of the concrete behavior view is very simple, including only two iterative steps as shown in Figure 24. The first one is to define the inner behavior of capsules that are identified and structured in the concrete structural view. The second one is to refine conceptual collaboration scenarios for capsule interactions i.e. sequence diagrams.



Capsules defined in conrete structural view
Conceptual collaboration scenarios
1. Capsule inner behavior
2. Capsule interactions

Figure 24. Design steps of concrete behavior view.

At the end of concrete behavior view design, the system behavior is represented through state machines and message sequence charts. The design steps are next described in detail.

Step 1: Define inner behavior of capsules

In this step, the puzzle, what happens inside the capsule is to be solved (Figure 25). The solution is modeled with a state machine that is always associated with a capsule.

What is known in advance and taken as the input for this step, are the in-signals and out-signals defined in the concrete structural view. It has to be figured out, what states are needed, which signals trigger the state transformations and what are the concrete actions attached to states and transformations.

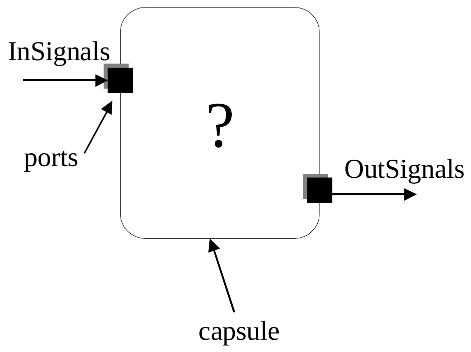


Figure 25. Capsule inner behavior is to be defined.

States may also contain sub-diagrams, or other state machines that represent different hierarchical state levels. The receipt of a signal event triggers the capsule's state transformations. When a capsule receives a message from another capsule, a signal is generated and some response by the capsule is usually required.

Step 2: Refine collaboration scenarios as capsule interactions

In this, collaboration scenarios identified in the conceptual behavior view are refined in forms of message sequence charts. Message sequence charts are defined for these specific scenarios' illustrating the message sequences between concrete architectural components i.e. capsules in the system.

It is very likely that protocol signals identified in structural design are not complete. Possible gaps are found out in behavior design.

7.3 Concrete Deployment View

Architectural elements, *language* and *design steps* are next described for the concrete deployment view. *Architectural element* modeling is concretized with the concrete deployment *language* and used in architectural descriptions of the concrete deployment view. Design is done with the following *design steps* that are defined in 7.3.3.

7.3.1 Architectural elements

The concrete deployment view is used to record the following architectural elements: concrete deployment hardware and software components, relationships between hardware components and relationships between software and hardware components (Table 9).

Concrete hardware components represent physical devices, more specifically, a computational resource having memory and processing capability. Hardware components may represent either type: *node* or *device*. Component instances reside and run on concrete hardware components. Use hardware components to model the topology of the hardware on which your system executes.

Software components are used to model the physical elements that may reside on a hardware component, such as executables, libraries, source files, and documents. The deployment software component, therefore, represents the physical packaging of the logical elements, such as capsules and protocols.

Table 9. Concrete deployment design elements.

Element	**Type of an element**	**Responsibilities of an element**
Concrete deployment *hardware* component	Node Device	What it has to do?
Concrete deployment *software* component	Component	How it does it?
Concrete deployment *hardware* relationship	Connection	Why? (Design rationale)
Concrete *hardware-software* deployment relationship	Allocation	

The most common relationship between hardware components is an association. In the context of deployment, an association represents a physical *connection* between hardware components.

The other relationship that is concerned in the concrete deployment view is the relationship between software and hardware i.e. allocation. The allocation relationship presents which software component instances will run on which hardware components.

7.3.2 Concrete deployment language

This sub section describes how concrete deployment is described using the Unified Modeling Language UML.

The deployment of concrete software component instances to hardware is described with a deployment diagram. It provides a basis for understanding the physical distribution of the run-time processes across a set of processing nodes. Nodes/devices may contain component instances, which indicate that the component runs on the node/device. *A deployment diagram* is a graph of *nodes/devices* connected by a communication association called a *connection* (see Figure 44).

7.3.3 Design steps of concrete deployment view

The design on the concrete deployment view includes the three steps shown in Figure 26. Concurrency of all steps is allowed. That is, the ordering of steps is relaxed and not strictly ordered.

The concrete deployment view considers deployment nodes that are defined in the conceptual level and concrete structural elements as input for this design phase. Inputs and steps together result in a deployment diagram. Steps are next defined in more detail.

Step 1: Package capsules and protocols to components

In the context of concrete deployment, a component represents the physical packaging of the logical elements, such as capsules and protocols. A concrete component can have instances and specific component instances can be assigned to nodes (Figure 27). The mapping from capsules to components may be one-to-one or one component may include several capsules and also refer to the communication protocols used between those capsules.

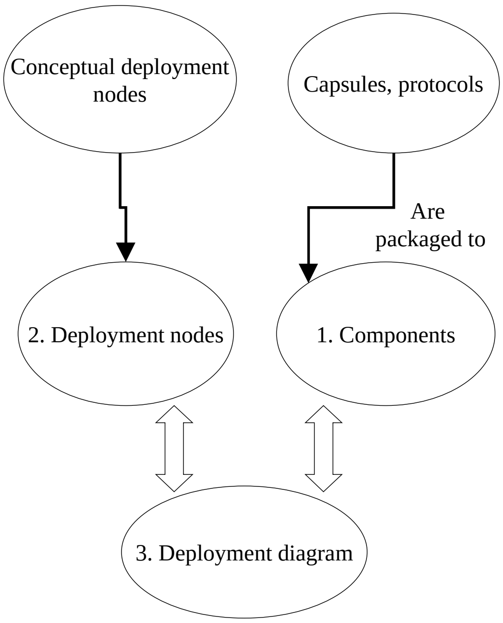


Figure 26. Design steps of the concrete deployment view.

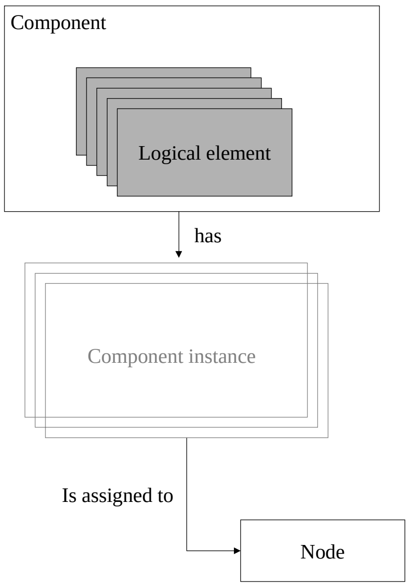


Figure 27. A component in the context of concrete deployment.

Step 2: Define deployment nodes

Nodes are used to model the topology of the hardware on which a system executes. The most common relationship between nodes is an association. In the context of deployment an association represents a physical connection between nodes. Concrete deployment nodes may be equal to conceptual deployment nodes or they may be refined or modified.

Step 3: Build up a deployment diagram

The final result of the concrete deployment view is a deployment diagram that is finally built up utilizing the hardware and software components defined earlier.

Figure 27 graphically illustrates how the deployment diagram is built. Logical elements e.g. capsules are packaged into components (Step 1) and after that component instances are allocated to nodes forming a deployment diagram.

8. Quality analysis of the concrete architecture

This section describes how the quality analysis of the concrete architecture can be done with quality analysis that is based on customer value analysis and with scenario-based quality analysis.

8.1 Quality analysis based on customer value analysis

One of the purposes of the architecture evaluation is both to assess the quality of a given architectural design relative to the design goals, and to quantify the measure of quality so that different designs can be meaningfully compared. Domain architecture analysis (Figure 28) could be done using the results of domain suitability and commonality analysis performed during product line scope (see Figure 8).

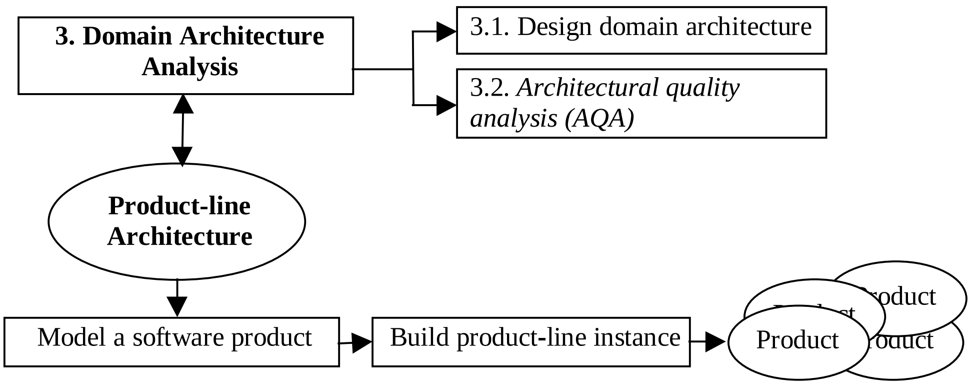

3. Domain Architecture Analysis
3.1. Design domain architecture
3.2. *Architectural quality analysis (AQA)*
Product-line Architecture
Model a software product
Build product-line instance
Product
Product
Product
Product

Figure 28. Product line architecture analysis integrated in domain architecture analysis.

Customer value analysis results may be used both to identify which changes are most important and to quantify the expected return on making the change. The change scenarios are used against the current architecture to determine the expected cost of evolving a product without using a product line approach. It can then assess the costs and benefits of each approach based on the value of the product changes the customer wants and the costs of making such changes under each development paradigm.

As it is described in Figure 28, architecture analysis is integrated in the domain modeling activities that have the result of product-line architecture and software products based on the developed product-line architecture. Here is considered a compositional approach to domain modeling in which a common, reusable architecture for the product line is developed. The generation of various members of the product-line can be done using adaptable, parameterized components (e.g. as in the CelsiusTech architectural case study [9]). As part of the domain modeling activity, the architecture must be evaluated against its quality requirements. In particular, a detailed, quantitative evaluation of how well the architecture instantiates the commonalties and accommodates the variabilities that characterize the product-line is desired.

A detailed and quantitative analysis is developed by:

- Creating scenarios based on the results of commonality analysis and

- Evaluating them based on the results of CVA and an analysis of the cost and benefit associated with potential variations in the scope of the family.

One or more scenarios are created for each commonality and each variability point. The cost and effort of the evaluation can be adjusted by using the value metrics to order the possible variabilities according to relative value and creating scenarios only for those variations above some threshold value.

The scenarios are tested against the architecture using qualified personnel to determine whether and to what extent the candidate architecture must be changed to accommodate each scenario. Where accommodating a scenario would require a change in the architecture, the expected cost to make that change is estimated. In general, scenarios developed from the commonalties should result in no architectural changes. Scenarios developed from variabilities within the product line scope should result in changes that can be accommodated by the variability mechanism (e.g. parameterizing a module or substituting one module for another).

Overall evaluation of the architecture is carried out using customer value data to assign weights to scenarios and scenario interactions. This weighting can be used to evaluate one candidate architectural design against another.

8.2 Scenario-based quality analysis of the concrete architecture

In order to assess the quality of a concrete architecture we use the software architecture analysis method only because, due to the revolutionary initiation approach, we are not able to perform customer value analysis (this can be done when the first version of the new concept is ready). As stated above this is a scenario-based method that consists of formulating a number of scenarios and evaluating the effect of each of them on the architecture. A scenario is a description of an expected use of a specific product line.

Generally, an analysis method can be developed to compare two or more candidate architectures or to assess the quality of a single architecture. Our analysis method is used to assess the quality of a single architecture, namely product-line architecture, that represents the commonalties and encapsulates variabilities of product-line members. The steps of the method are introduced next.

Step 1: Derive change categories based on product-line scope

Two categories of change are depicted in Figure 29. The first category contains scenarios that are related to the technical requirements of the platform. Scenarios of this category explore the applicability of the PLA in situations with various technical requirements. The second category of scenarios concentrates on context or environment identification and is evaluated with scenarios that simulate changes in the technical environment.

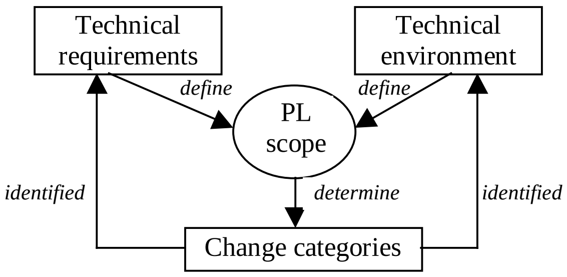


Figure 29. Derived change categories.

Step 2: Scenarios identification

The second step of the method consists of the scenario's identification. In this activity we distinguish possible changes that may happen in the life of the PL based on the derived categories. Scenarios should illustrate the kinds of anticipated changes that will be made to the PLA due to the PL scope. For example, from changes in technical environment (the quality requirement of portability) we can derive the following scenario:

What happens when another network protocol is to be used?

By formulating a number of these scenarios, we can make portability tangible, because they capture what we actually want to achieve with portability.

A common problem of the scenario identification activity is when to stop generating scenarios. Possible solutions could be suggested:

- The set of scenarios is complete when the addition of a new scenario no longer perturbs the design.
- Try to identify a complete set of scenarios, but this is generally impossible.
- Delimit a representative set of scenarios, which has the weak point in how to define which is the representative set. This solution is based only on the creativity and subjectivity of the analyst, or it requires a domain knowledge base organized in the three important sets characterized previously.

- Consider various criteria for the relevant scenarios. For example, scenarios which are possibly complex to realize. A two-dimensional framework plan (categories of complex scenarios, sources of changes) may help identify complicated scenarios.

- Try to apply a procedure for identification. For example, a two-step procedure, where in the first step, a coverage guarantee is obtained. The scenarios are identified and clustered, based on the objectives and domain experts' knowledge, and the coverage is checked against the objectives of the stakeholder, architecture and quality. The second step validates the balance of scenarios with respect to the objective, based on a Quality Function Deployment (QFD) technique [58]. The decision to distinguish more scenarios is made based on comparison against 1 (one) of a calculated imbalance factor for each quality attribute.

- Use a set of standard quality attribute-specific questions to ensure the proper coverage of an attribute by the scenarios. The boundary conditions should be covered. A standard set of quality-specific questions allows the possibility of extracting the information needed to analyze that quality in a predictable, repeatable fashion.

We assume that the architecture is a good one and it is not necessary to generate scenarios to verify the functional requirements. Otherwise these should also be considered when verifying functionality.

Step 3: Description of the PLA

Another required activity is the *description of the PLA.* This activity is considered the third step, but it could be performed in parallel with the previous one. The method is applied to analyze the architecture, when not all the necessary structures have been designed and the architectural representation might not include some concrete domain requirements.

A specific of PL architecture is that it contains abstractions of the problem domain but also concrete components, which could be common or variants of different family products. PLA combine the generics of the domain with the concreteness of each product and its views must reflect the diversity and the range of this diversity through mandatory, optional and alternative representations. The minimum of required views for a PLA representation are considered from the QAD method.

Step 4: Evaluate the effect of the scenarios on the architecture elements

The evaluation of the scenarios effects on the analyzed PLA view may consider several issues. We can identify some of these in the following enumeration:

- A classification and generalization of the architectural elements facilitates the estimation of cost or effort required for changes to be made. Determine if the architectural elements influenced are members of abstract or concrete PLA packages.

- The effect of a scenario is estimated by investigating which architectural elements are affected by that scenario. The cost of the modifications associated with each change scenario is predicted by listing the components and the connectors that are affected and counting the number of changes.

- The evaluation can be performed in a complete manner, if the set of identified scenarios is complete. If all scenarios are executed without problems, the quality attribute of the architecture is optimal.

- The evaluation can be performed in a statistical manner, if a representative set of scenarios has been considered. The ratio between scenarios that the architecture can succeed with and scenarios not succeeded with well by the architecture provides an indication of how well the architecture fulfills the software quality requirements.

- In case the analysis is performed at a time when PL has been already developed and multiple releases exist, it is possible to define and use a measurement instrument to express the effect of scenarios. The instrument must indicate, not only the impact of a scenario considering both the flexibility in space (multiple variants in products) and time (multiple versions of variants), but also whether multiple owners are involved. The analysis of flexibility in time could indicate whether it leads to version conflicts.

The objective of evaluation is to get a prediction of the quality of the PLA with respect to the anticipated variability in functional or non-functional characteristics of this product line.

To evaluate the effect of these scenarios on the architecture, we classify the effect of a scenario into four discrete levels:

- At the *first level*, no changes are necessary, which means that the scenario is already supported by the architecture.

- At the *second level*, just one component of the architecture needs to be changed. At this level, we have true locality of change.

- At the *third level* more than one component is affected, but no new components are added or existing ones deleted. This means that the structure of the architecture remains intact.

- At the *fourth level* architectural changes are inevitable, because new components are necessary or existing ones become obsolete.

It is clear that one should seek to keep the level of effect as low as possible.

When we return to our example scenario, we see that this scenario necessitates a change in the *CommunicationService* component. Thus, this scenario has a level two effect. This means that we have locality of change for this scenario and that the architecture is portable with respect to the network protocol used.

Step 5: Scenario interaction

The result of the effects evaluation may represent the input for this last step where the *scenario interaction* is revealed. The activity is to determine which scenarios interact, meaning that they affect the same component. High interaction of unrelated scenarios could indicate a poor separation of concerns.

9. Guidance to apply the QADA method

From the method user's point of view, the guidance is an essential part of the method. Here, a case study is used as guidance to apply the QADA method. The case study is a revolutionary initiation of a new product line for distributed service platforms (DiSeP). Revolutionary initiation of a new product line means development of a complete product-line architecture and set of components before developing the first product in a new domain [12].

9.1 Domain vocabulary of the case study

Modern distributed systems are software-intensive systems that embody service architecture, thereby providing a variety of services for their users. The 'service' and 'user' terms are overloaded, and therefore, it is essential to define the elements of service architecture and their meanings. The QADA method applied here for the conceptual and concrete architecture of a distributed service platform uses the following concepts as basic elements of service architecture.

By a *platform* we mean here an implementation of a distribution platform, formed by executing *modules* in a networked environment. The *modules* are built by following the architecture of the platform.

A module is considered here as an independent program running in a restricted processing environment as in a centralized operating system, in a *device* or in a virtual machine. A *device* is an independent electronic entity with processing capability and the ability to store and transfer data.

The module consists of at least two parts: The first part, which takes care of the interoperability between the *modules* by executing the operations of the *platform* and the second part that consumes the benefits of the platform, as a *user* or as an *application service*. *A user* is an entity that utilizes the *services* provided by a module. An entity may be, for example, an operationally restricted entity of an application, a device or a user interface.

A service is an abstract sequence of operations that may include storing, processing, retrieval, deletion, comparison and classification of new or existing data. The use of a *service* either leads to the success of a desired operation or to a failure. There are three kinds of *services* considered here: *basic system services, system services and application services*.

Basic system services are a default pack of services and are produced by the platform. *System services* are also produced by the platform with the difference that system services utilize the *basic system services. System services* can be operated by the applications through special interfaces. The last group into which the services are classified, is the application services group, which is separate from the *platform*. *Application services* are distributed over the different *modules* of a *network* by using the *platform*.

By the term *a network* here is meant an aggregate of two or more *modules* and *connections* between them. *A connection* is the ability to transfer information from a *module* to another. In *a connection*, the receiver of the information is always capable of resolving the way to reach the sender and *connections* are maintained internally by the *modules* following the needs of the *platform*.

9.2 Requirements engineering

The requirements engineering phase of DiSeP follows. This phase defines the context and includes the requirements specification and the requirements analysis.

9.2.1 Defining context

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 utilizes the platform (Figure 30). 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 e.g. the number of modules in Figure 30 or the range of the available services, 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.

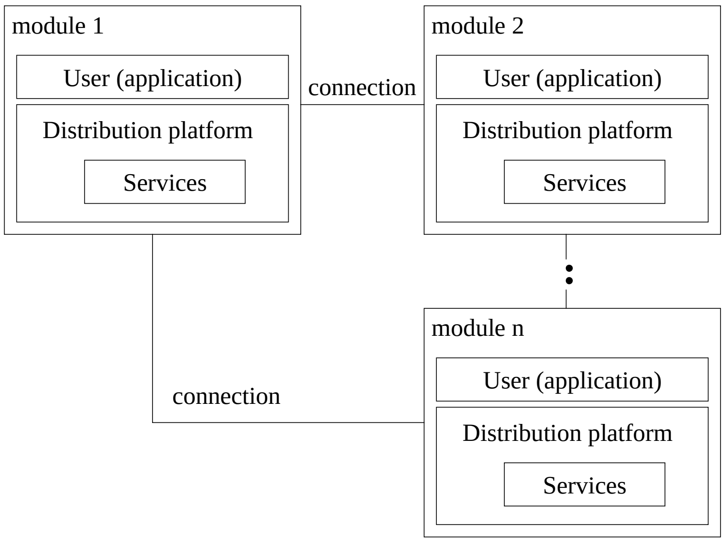


Figure 30. A network consisting of a distribution platform and its context.

The interoperability is reached by defining certain essential services (i.e. services that are called system services) *mobile.* A *mobile service* means that the current state and the data contents of a service can be transferred from one module to another. Thus the service in question becomes *passive* in the first module and *active* in the another module.

9.2.2 Requirement specification

Technical properties i.e. the main goals in the system development are:

1. Platform implementation independence. Independence of implementation languages between modules is provided by eXtended Markup Language XML technology [14] as a universal communication means.

2. Distribution transparency. The network is able to resist dynamic changes in the configuration of the network of interconnected modules or in the physical communication links between different devices.

3. Mobility of system services. System services are not centralized in one location, but any of the units can act as a system service provider.

9.2.3 Requirement analysis

The first technical property denotes independence of implementation languages and the use of a universal communication manner between modules, i.e. portability. The second property denotes the ability to resist dynamic changes in the configuration of a network of interconnected modules or in the physical communication links between different devices.

The third property means that system services are not centralized in one location, but any of the modules can act as a system service provider.

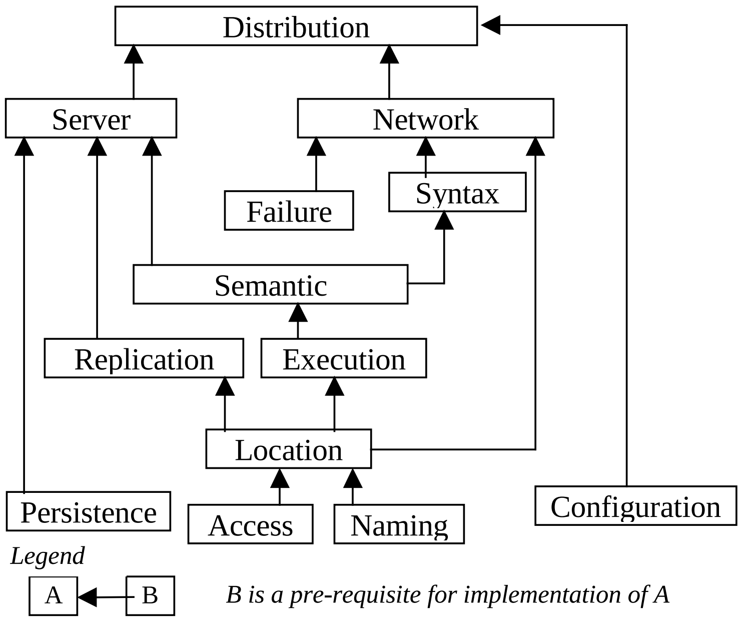


Figure 31. Prerequisites for fulfilling distribution transparency.

The last two means distribution transparency [2], i.e. transparency of the network topology, services and migration (Figure 31) that can be considered as the adaptability of the platform at the architectural level.

9.3 Designing conceptual architecture

The conceptual architecture of the case study is designed next using the three architectural views and design steps defined for each view. The next sections also provide a window into the architectural documentation of a distributed service platform.

9.3.1 Conceptual structural view

When considering the structural viewpoint of the service architecture, we started with the first step: defining the functional responsibilities for the platform. This design phase was iterative with other two steps: defining the non-functional responsibilities and decomposing the system. The final results are shown on the next few pages. These figures present the completed documentation of this view.

Step 1: Define functional responsibilities

The lists of functional responsibilities are introduced next. Figure 33 illustrates the functional responsibilities of the platform user interface. The user interface is divided into four interface components that will take the form of the conceptual leaf components and subsystems in Figure 35. Every one of these components provides the user with the possibility to utilize different services which are located in a local module or in the distributed modules of the platform.

Figure 32 shows the functional responsibilities for a subsystem component "system services". This subsystem consists of two conceptual leaf components: Lease service and Directory service. These system services are accessed through the four interfaces described in Figure 33.

Both the services have two interfaces, which implement the different roles the user application may express. For example, the user application may access the directory service in two different roles: as an *application service user* or as an *application service provider*.

System services (subsystem component)
- Produce *services* that are not autonomous but activated by the autonomous parts of the platform
  - **Lease service (leaf component)**
  - Utilise lease management between two independent *units* or other logical elements:
    - Accept and host leases of lease grantors
    - Grant leases of lease grantors for lease users
    - Take care of lease renewals for any leased system resource
    - Keep track of lease renewals for any shared and leased resource
  - **Directory service (leaf component)**
  - Provide a directory service interface to the distributed data storage
    - Register and unregister service proxies
    - Keep track on registered services
    - Search for requested services
    - Send a requested proxy for the user

Figure 32. Functional responsibilities of the "system services" subsystem.

System service user interface (subsystem component)
Provide interface for services that are directly accessible by the user
- **Application service user (leaf component)**
- Allow *users* to use application *services* through a directory service interface:
  - Allow application to search for suitable *services*
  - Allow application to fetch a *service* proxy
- **Application service provider (leaf component)**
- Allow *users* to provide application *services* through a directory service interface:
  - Allow application to create an appropriate service proxy
  - Allow application to register the service proxy to a directory service
  - Allow application to unregister the service proxy from the directory service
- **Lease user (leaf component)**
- Allow user to (re)negotiate for a lease with the provider of the desired service
- **Lease grantor (leaf component)**
- Allow user to grant lease(s) of provided service(s)

Figure 33. Functional responsibilities of the ”user interface” subsystem.

The last subsystem contains the basic system services, which perform all the functions (Figure 34) that are essential for a properly working distribution platform. The “Basic system services” subsystem produces controlling, advertising, messaging, data distributing, observing and data transferring services. Components inside the “basic system services” subsystem serve each other or the system services on the upper layer of the platform module.

Basic system services (subsystem component)
Produce *services* that operate autonomously
Control service (subsystem component)
Control services
Control service (leaf component)
Monitor the state of the network
Activate system services
Track that the need for redundant data copies is satisfied
Negotiate about the copying, transferring or deleting the data if necessary
Interpreter service (leaf component)
Location services (subsystem component)
Data distributing service (leaf component)
Contribute to the operation of a distributed data storage
Maintain, track and create *connections* to other *units* in order to share data
Redundantly copy data between different *units* in order to improve robustness and fault tolerance of the system
Allow data to be stored in local resources
Location service (leaf component)
Locate a system *service* provider
Listen for beacon signals and keep track of the system *service* providers through them
Hide the sender address of incoming advertisements
Announce the availability of the system *services*
Send beacon signals about the location and the state of the system *service* provider
Messaging service (component)
Provide messaging *services*
Create a mailbox through which the internal *services* of the system may communicate with each other in an asynchronous manner
Receive and host mailbox messages until they are fetched (by system services)
Act as an outbox for all (both mailbox and direct) messages
Observing service (component)
Route direct messages from the network to registered listeners
Forward mailbox messages to Messaging Service
Data transfer service (component)
Act as a data producer of the observing *service*
Comprise an up-to-date list of data transfer blocks in other *units*
Communicate with data transfer blocks in other *units*
Receive data from sending Data Transfer Block (DTB) and pass it to the observing *service*
Send data from the messaging service to the receiving DTBs (in other *units*)

Figure 34. Functional responsibilities of "basic system services" subsystem.

Step 2: Define quality responsibilities

In addition to the functional responsibilities the non-functional quality responsibilities have also to be defined. Functional responsibilities are listed in Table 10. Responsibilities are categorized according to quality attributes. This phase does not try to define quality requirements for the whole system but to encapsulate each requirement as a responsibility of a restricted part of the architecture in order to localize the styles that will be selected in Step 3. Passive components related to adaptability in Table 10 refer to dynamic architecture structure specific for the case study.

Table 10. The table of non-functional responsibilities (NFRs) of the system.

Quality attribute	**Requirement**	**Who/How?**
availability	System services are always active in one of the *units* in the *network*. Services are activated and shut down under control.	Control Service
reliability	The content of the data (storage) is granted to be same in every replicate. Even after a failure of hardware or software around the system.	Data Storing Service
adaptability	Distributed platform can be used as a platform for applications running in diverse end point devices.	Architecture contains both optional and passive components
portability	Platform supports diverse network protocols.	Communication Services
extensibility	Components (at least system services and some components inside Communication services) may be added or changed after platform execution has commenced.	Components obey certain policies, conventions, and protocols (design rationale) to ensure interoperability among updated or added components.

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 [13]. Change can be stimulated for instance by time or by user interactions. Optional components turn to passive or active while moving from static dimension of architecture into dynamic ones.

Step 3: Cluster responsibilities to components and subsystems

The conceptual structural view includes conceptual subsystems and leaf components. Grouping responsibilities in a textual list (Step 1) mainly forms these design elements. In addition, the quality attributes defined in step 2 drive the selection of architectural styles and thus affect the forming of the system architecture.

Table 11 contains four styles used in the platform. A layered style, like all styles, reflects a division of the software into units. Each layer represents a collection of software that together provides a cohesive set of services [3]. Other software can utilize these services without knowing how those services are implemented. A layered style achieves the qualities of portability, mobility and adaptability, because if the layering is pure, e.g. *porting* the system involves only the re-implementation of the portability layer. Layers also support limiting adaptation, required due to changes, to the part of software responsible for the new properties of systems.

The main characteristic of the blackboard style is that shared data is in the blackboard, i.e. in memory, and the world outside (clients) responds to changes in the blackboard. A blackboard is an active blackboard if it sends notifications to subscribers when data of interest changes. The blackboard style achieves the qualities of reliability and availability. *Reliability* is achieved through data replicas. Because the clients are independent of each other and the data store is independent of the clients, new clients can be easily added i.e. the system is *scalable*.

The third style, *implicit* invocation, means that objects announce (i.e. publish) events via *multicast* and these events invoke the associated procedures in subscribers. This style supports availability and extensibility.

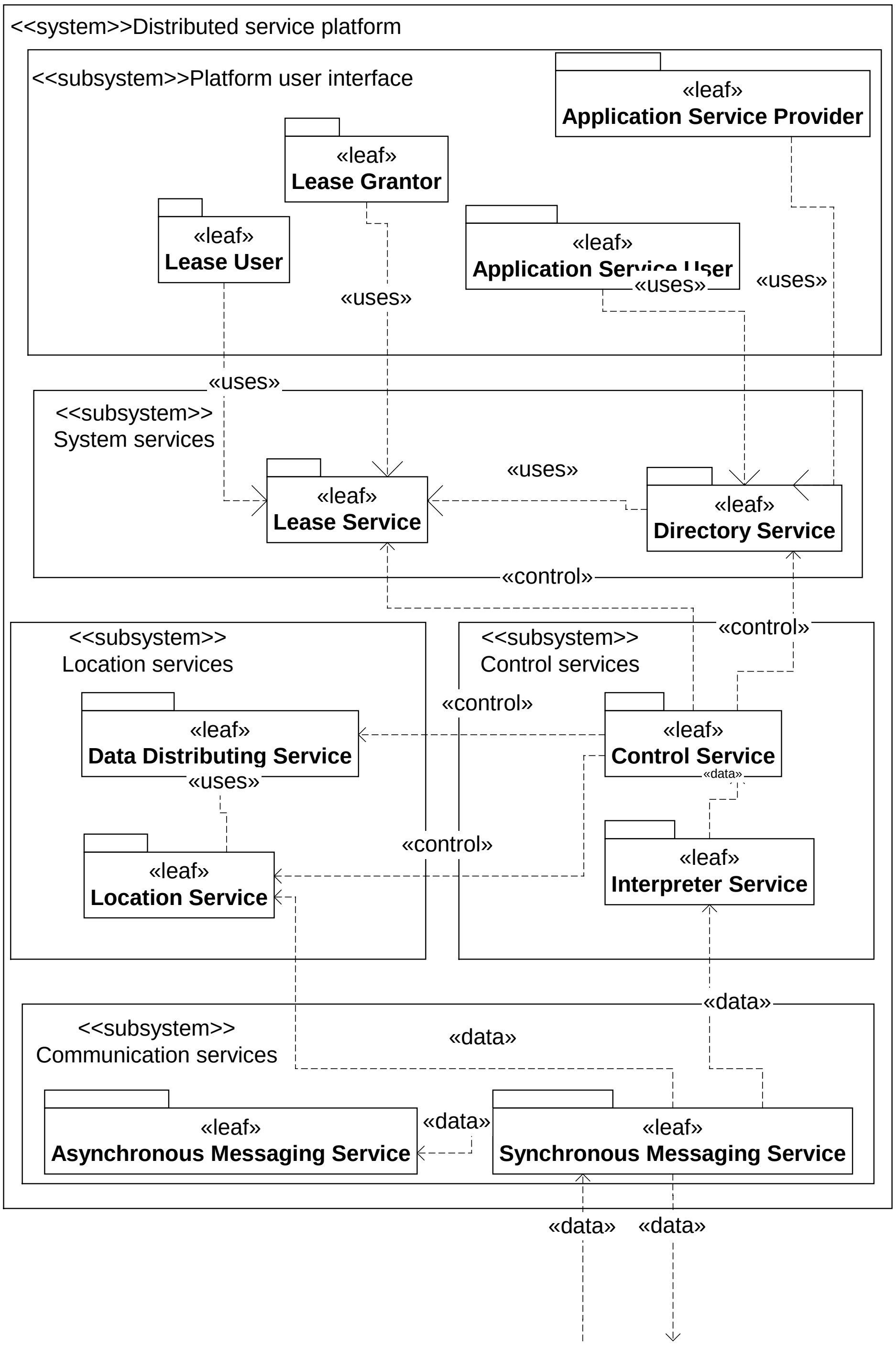
In the object-oriented architectural style objects provide interfaces to access services and services are invoked by calling interface methods [9], though services can be easily modified supporting *modifiability*. And if and when the interface methods are known, the objects utilizing services through methods can be easily added i.e. means *scalability*.

Table 11. Styles, design rationale and quality attributes supported by styles.

Style	**Design rationale**	**Quality attribute**
Layering	A layer provides a 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, scalability
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, scalability

A decomposition model showing the hierarchical structure and conceptual relationships between components is seen in Figure 35.

Basic system services, which are here considered to be location, control and communication services are produced by the platform. System services are also produced by the platform with the difference that system services are utilizing the basic system services. System services are operated by the applications through interfaces and through these system services, applications can distribute application services over different *modules* in a network.



<<system>>Distributed service platform
<<subsystem>>Platform user interface
«leaf» Application Service Provider
«leaf» Lease Grantor
«leaf» Lease User
«leaf» Application Service User
«uses»
«uses»
«uses»
«uses»
<<subsystem>> System services
«uses»
«leaf» Lease Service
«leaf» Directory Service
«control»
«control»
<<subsystem>> Location services
<<subsystem>> Control services
«control»
«leaf» Data Distributing Service
«leaf» Control Service
«data»
«uses»
«control»
«leaf» Location Service
«leaf» Interpreter Service
«data»
<<subsystem>> Communication services
«data»
«data»
«leaf» Asynchronous Messaging Service
«leaf» Synchronous Messaging Service
«data»
«data»

Figure 35. Conceptual structural model of the service platform.

9.3.2 Conceptual behavior view

Below, the behavior view of service architecture is designed following the three steps defined for this view. The first and the second step are introduced in parallel by courtesy of the final document that is the common result of both steps. Step 3 creates a collaboration model based on the results achieved in steps one and two.

Steps 1 and 2: Select collaboration scenarios and identify service sets

Abstract descriptions of collaboration i.e. *collaboration scenarios* are shown in Table 12 and Table 13. Scenarios are derived from functional responsibilities and aimed to cover the main co-operation situations between the user applications and the system and also inside the system. The main using scenarios and service sets related to them, are scenarios from one to four in Table 12. These scenarios describe the situations when a user application accesses the services through interfaces. In addition to the scenario descriptions, the components participating in these scenarios are also specified. Mentioning the component in parenthesis means that the necessity of the component in question is dependent on the current state and structure of the dynamic platform architecture.

Table 12. Collaboration scenarios when using the service platform.

Scenarios:	**Services participating the scenario(s) in question**
1. Application searches for suitable *services* 2. Application fetches a *service* proxy	Application service user Lease user Lease service Directory service Data distributing service (Communication services)
3. Application registers the service proxy to a directory service 4. Application unregisters the service proxy from the directory service	Application service provider Lease user Lease grantor Lease service Directory service Data distributing service (Communication services)

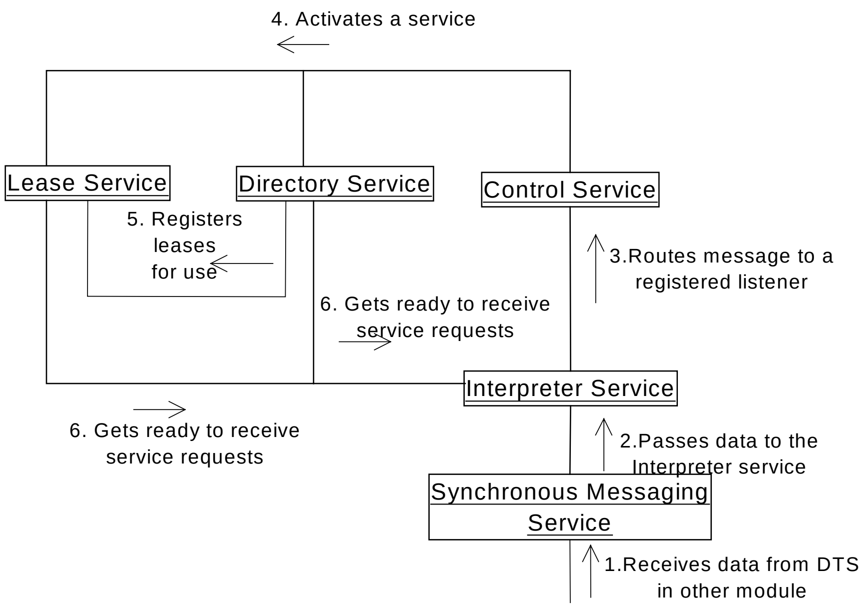
In addition to these scenarios, there are still a few important collaboration cases occurring among the autonomous parts of the platform. These scenarios and service sets are specified in Table 13.

Table 13. Collaboration scenarios among autonomous parts of the platform.

Scenarios:	**Services participating the scenario(s) in question**
5. Controlling service activates system services 6. Controlling service deactivates system services	Synchronous messaging service Controlling service Lease service Directory service
7. Advertising service creates an ad and broadcasts it.	Synchronous messaging service Data Distributing Service

Step 3: Create collaboration model

Textual descriptions lead to graphical collaboration descriptions when drawn up into diagrams. Each scenario is illustrated as a *collaboration diagram* representing an ordered sequence of actions, which is a refined collaboration scenario, and the service components related to that sequence. As an example, Figure 36 shows a collaboration diagram for one scenario. This scenario describes the situation where the controlling service activates system services (Lease Service and Directory Service).



4. Activates a service
Lease Service
Directory Service
Control Service
5. Registers leases for use
3.Routes message to a registered listener
6. Gets ready to receive service requests
Interpreter Service
6. Gets ready to receive service requests
2.Passes data to the Interpreter service
Synchronous Messaging Service
1.Receives data from DTS in other module

Figure 36. Collaboration diagram for the scenario "Controlling service activates system services".

9.3.3 Conceptual deployment view

The deployment view of the conceptual architecture is done with the three steps described next. At first, *leaf components are clustered to units of deployment*. Secondly, *computing nodes in the system are identified*. The third step is to combine deployment nodes and units of deployment with *allocation* relationships.

Step 1: Cluster leaf components to units of deployment

Conceptual components are grouped into units of deployment according to the mutual requirement relationships between components. Figure 37 shows how and why three components included in the "application service provider" deployment unit have a mutual requirement relationship with each other.

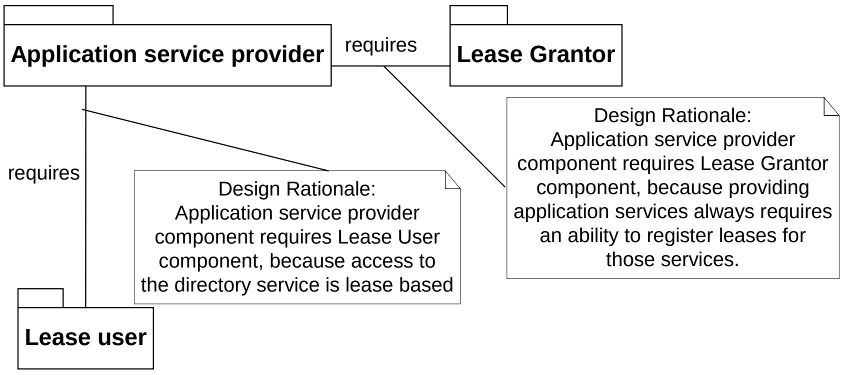

Application service provider
requires
Lease Grantor
Design Rationale:
Application service provider component requires Lease Grantor component, because providing application services always requires an ability to register leases for those services.
requires
Design Rationale:
Application service provider component requires Lease User component, because access to the directory service is lease based
Lease user

Figure 37. Mutual requirement relationship between leaf components inside the ”Application Service Provider” unit of deployment.

Other units of deployment are constructed similarly, always respecting the mutual relationships between components. The final clustering is shown in Table 14.

Table 14. Grouping conceptual components into units of deployment.

Leaf components	**Unit of Deployment**	**Type**
Synchronous messaging service Control service Data Distributing service Interpreter service Location service	Basic system services	Mandatory
Directory service Lease service	System services	Optional
Application service user Lease user	Application service user	Optional
Lease Grantor Application service provider	Application service provider	Optional

Table 15 provides the design rationale for clustering leaf components to units of deployment and selecting types for created deployment units.

Table 15. Design rationale for clustering units of deployment.

Unit of Deployment	**Design Rationale**
Basic system services	These services form the basis of the platform and are therefore needed in every deployment node.
System services	This unit is also needed in every deployment node, but is separated to its own deployment unit, because these services are active only in one node in the network *at a time*.
Application service user	**This specific deployment unit is needed if an application *uses* services of other applications**.
Application service provider	This specific deployment unit is needed if an application has services to *provide* for other applications.

Step 2: Identify deployment nodes

A deployment node is defined to be *a device* with single processor. *Nodes* may differ in processing capacity and they connect to each other spontaneously building up a network. In a spontaneous network, services provided by various applications come and go. Services present are registered in a special unit of deployment that is mandatory in each *node*, but is dynamically activated only in one *node* at the time.

Step 3: Allocate deployment units to nodes

Figure 38 depicts the logical alternatives to allocate deployment units of the service platform to physical nodes in the network.

Basic system services and system services are mandatory units of deployment in the platform with the difference that system services have a dynamic nature. A dynamic nature is illustrated with a special fill pattern.

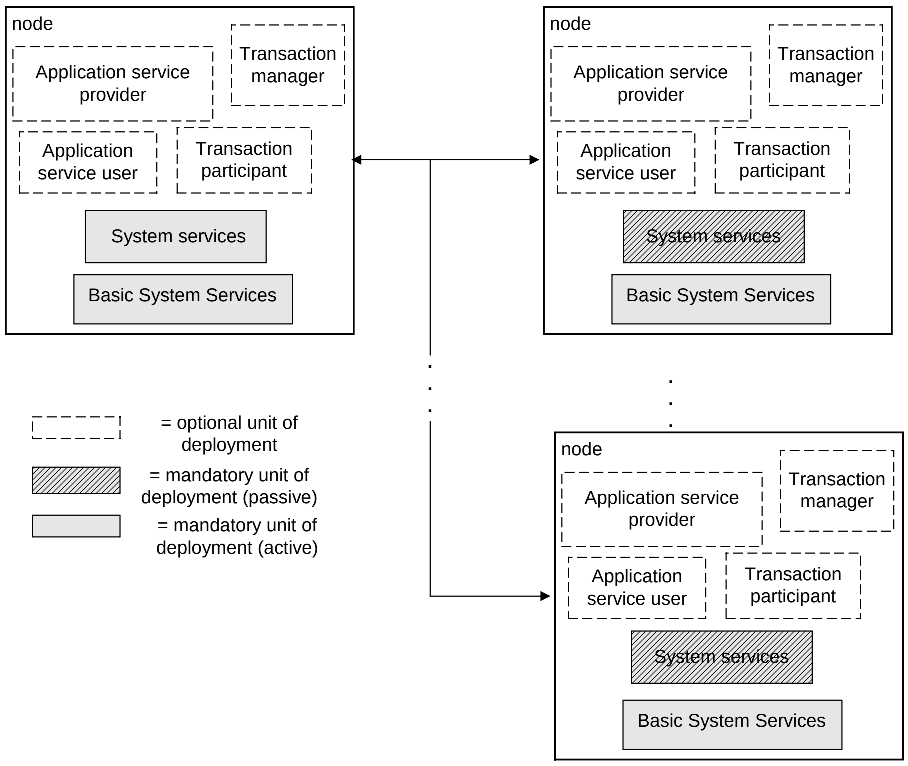


Figure 38. Enabled conceptual subsystem deployment.

9.4 Analyzing conceptual architecture

Analysis of the conceptual architecture focuses on variability analysis and identifying architectural styles and pattern violations, and therefore, control and communication mechanisms are under inspection.

9.4.1 Variability analysis and representation

Variability could be incorporated in each view of the architecture. For example, the structural view describes the components and their interconnection from the static point of view. On the conceptual level the variability could be expressed by showing what is variable or by using specific relations and notations (aggregation, specialization, etc.) of the modeling language (UML for example).

The usage of packages is a solution to express the diversity of specific subsystems and component models.

For the moment, in the architecture two variability points are included (see Figure 39), but others are implicit or unspecified. The upper level services inside the *SystemService* represent one of the variable points. In one of the product members there are two services: Lease Service and Directory Service that always come together. In other (future) products there also might be Security Service, Transaction Service, etc.

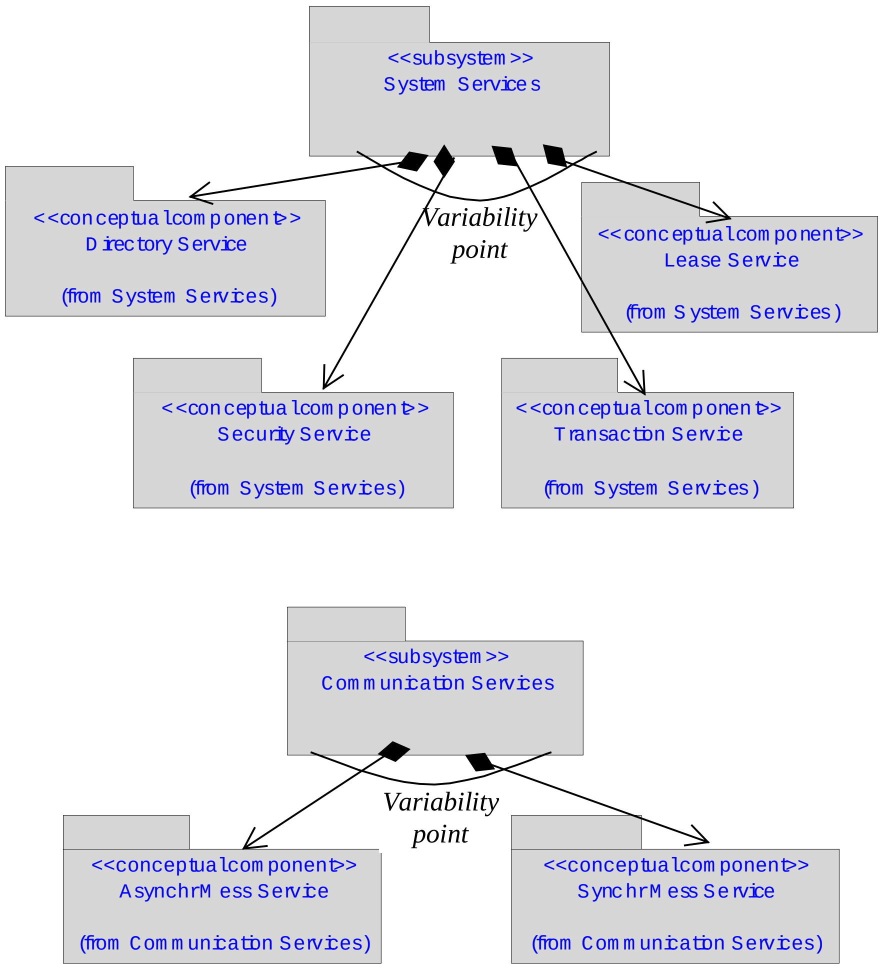


Figure 39. Variability points specified in the structural view of the architecture.

A second point of variability is in the lowest layer of the platform, inside the subsystem *CommunicationServices*. The communication could be performed either using *SynchronousMessagingService* or *AsynchronousMessagingService*.

9.4.2 Architectural styles and patterns violations analysis

We assume that at the conceptual level the attribute based methodology should be based on routine and predictability. Thus we have already defined and made it possible to extend a standard set of important attribute based analysis questions associated with each style.

Table 16 exemplifies the important features of the blackboard style for conceptual analysis. For instance, the blackboard's communication mechanism requires a single point of contact between the central Control Service unit and the other cooperative components, but the architecture that follows the style, in fact, has multiple points of contact under certain circumstances.

Table 16. Features to focus on the analysis of the blackboard style.

Feature	**Questions**
Control/ registration mechanism	When the blackboard wants to send a message to some units does it broadcasts the message to all units or simply sends the message to the registered units? Does the model support independent control or broadcast control? Is the control single threaded or multithreaded? Is the message control, data or both?
Communication mechanism	Is there a specific point of contact or multiple points of contact between the blackboard and the other units?
Violations	Are there any links that violate the control or communication policy?
Integrability and modifiability	If new components are added to the system, will they be integrated into the blackboard the same way as existing components?

Layer architecture style has also been considered for conceptual architecture design and Figure 40 presents these conceptual layers and the relations between them.

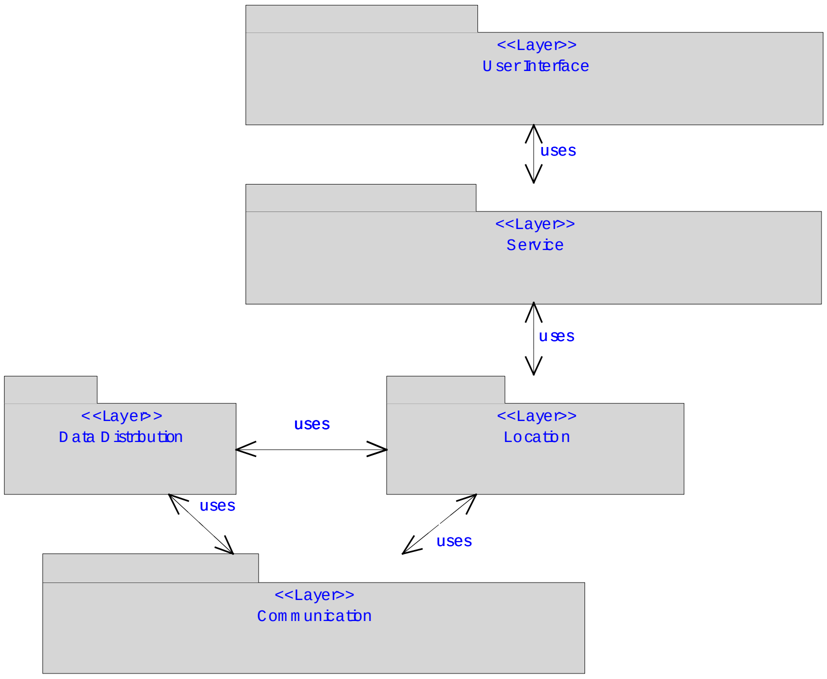


Figure 40. Conceptual architecture layers.

For the moment no violations of architecture styles or patterns have been determined. The accuracy of the analysis at this level depends on the question set and cooperation with domain experts.

9.5 Designing the concrete architecture

Concrete architecture descriptions of the case study are introduced next and design is illustrated using the three architectural views and design steps defined for each view. Architectural views are similar to conceptual views with the difference that they provide more lower level details of the system.

9.5.1 Concrete structural view

The design steps included to the concrete structural view design of the case study are discussed below.

Step 1: Refine functional responsibilities

In the case study, steps 1 and 3 were combined so that refined functional responsibilities were documented as a part of the structural diagrams. A separate document including capsule functional responsibilities and design rationale was then generated from the tool.

Step 2: Select design patterns

Table 17 shows as an example, how selected design patterns are documented in the DiSeP case study. The design pattern observer participates in the realization of three styles: implicit invocation, object oriented and blackboard style.

Table 17. Realizing styles through patterns.

Pattern	**Used times in the case study**	**Participates in the realization of a style**
Observer	3	Implicit invocation Object-oriented Blackboard

Step 3: Build a hierarchical structural diagram

The structural diagram of DiSeP includes several diagrams. Figure 41 illustrates a structure diagram for the subsystem capsule "system services" as an example. Services inside this capsule represent the *optional* capsule type whereas the container capsule is a *fixed* one.

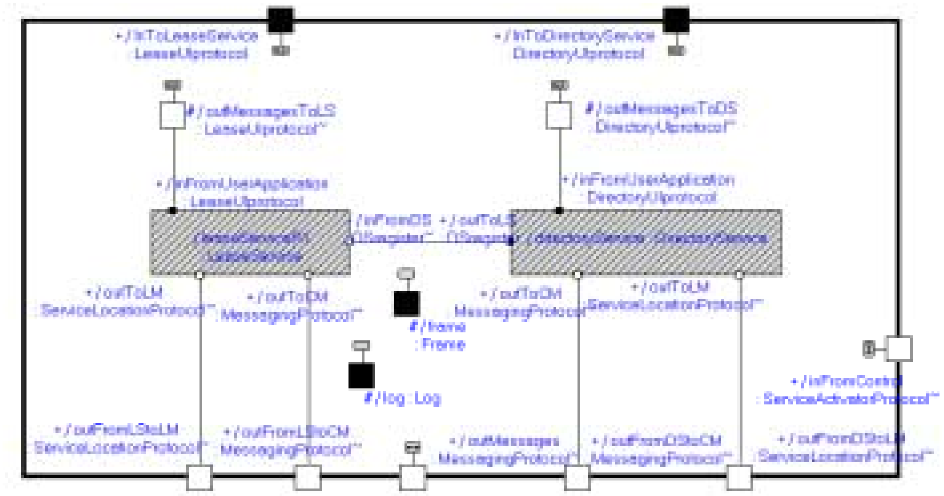


Figure 41. Structure diagram for the capsule "system services".

9.5.2 Concrete behavior view

The concrete behavior of the case study system is modeled and documented with state diagrams and message sequence charts. Next the design steps are discussed individually.

Step 1: Define the inner behavior of capsules

Figure 42 represents the inner behavior of the capsule “System service provider”. The capsule "System services" is initialized to achieve the "ready" state when the platform is executed. The optional contained capsules "LeaseService" and "DirectoryService" are not yet instantiated. Signals for activating and deactivating optional capsules come from the "Control" capsule. Service requests from the user application are handled in a choice point. If optional services are already instantiated, the request is fulfilled locally, but if optional services are shut down, the request is forwarded to the module where services are active (location data is managed by location service).

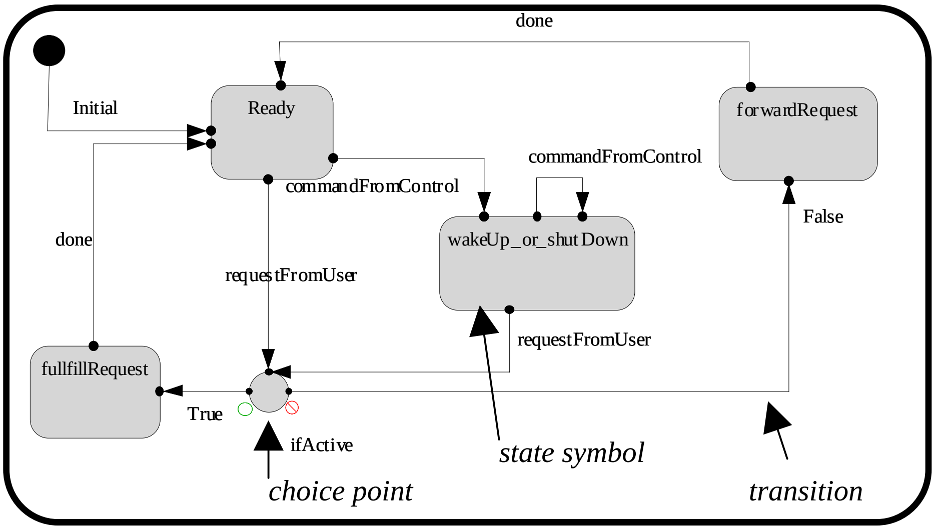


Figure 42. The top-level state diagram of the capsule "system services".

Step 2: Refine collaboration scenarios as capsule interactions

Another way to describe the behavior of a system, in addition to the behavior of individual capsules, is the *interaction* between a group of *capsules*. An interaction is a pattern of communication among capsules at runtime. Figure 43 shows concrete collaboration between capsules in a scenario that activates optional capsules. The same scenario is represented at the conceptual level in Figure 36.

An advertisement is received (1.1.1 notifyObservers) to capsule "Control" through capsules "ReceiveMethod" (1. incomingMessage) and "Communication-Manager" (1.1 XMLdataIn). The advertisement contains information that makes the control decide to activate the services in this module (1.1.1.1.1 and 1.1.1.1.2 invokeService). After being invoked, these services prepare to serve (1.1.1.1.2.1. registerLease and 2. addObserver).

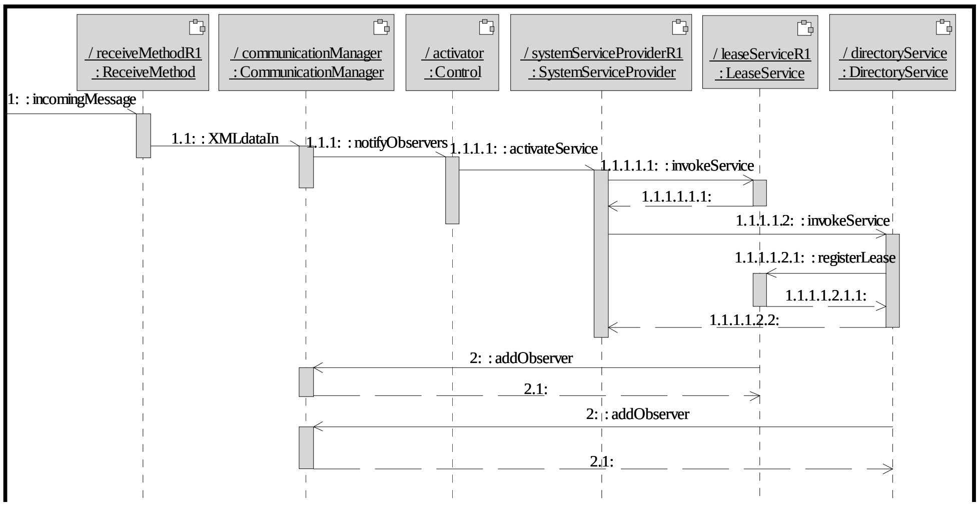

/ receiveMethodR1 : ReceiveMethod
/ communicationManager : CommunicationManager
/ activator : Control
/ systemServiceProviderR1 : SystemServiceProvider
/ leaseServiceR1 : LeaseService
/ directoryService : DirectoryService
1: : incomingMessage
1.1: : XMLdataIn
1.1.1: : notifyObservers
1.1.1.1: : activateService
1.1.1.1.1: : invokeService
1.1.1.1.1.1:
1.1.1.1.2: : invokeService
1.1.1.1.2.1: : registerLease
1.1.1.1.2.1.1:
1.1.1.1.2.2:
2: : addObserver
2.1:
2: : addObserver
2.1:

Figure 43. Sequence diagram for the scenario "activate system services".

9.5.3 Concrete deployment view

This section provides the design steps performed during the design of the DiSeP case study and presents a deployment diagram as a result.

Step 1: Package capsules and protocols to components

Packaging capsules to components was one-to-one mapping in the case study. Communication protocols related to each capsule were added by the tool in the packaging phase.

Step 2: Define deployment nodes

Deployment nodes defined for the concrete deployment view do not differ from those defined for the conceptual view. An issue that is refined, is the description language that decomposes *conceptual node* to a *device* and a *processor*.

Step 3: Build up a deployment diagram

The case platform has a dynamic architecture in a spontaneous network. This means e.g. that a platform that takes the role of system service provider is elected and that the number of available communication methods in the platform varies. These reasons lead to the situation that the configuration of DiSeP varies between different processors. For these reasons, it was difficult to build up a complete deployment diagram for DiSeP. Instead of a complete diagram, Figure 44 shows an example network consisting of two devices with single processors.

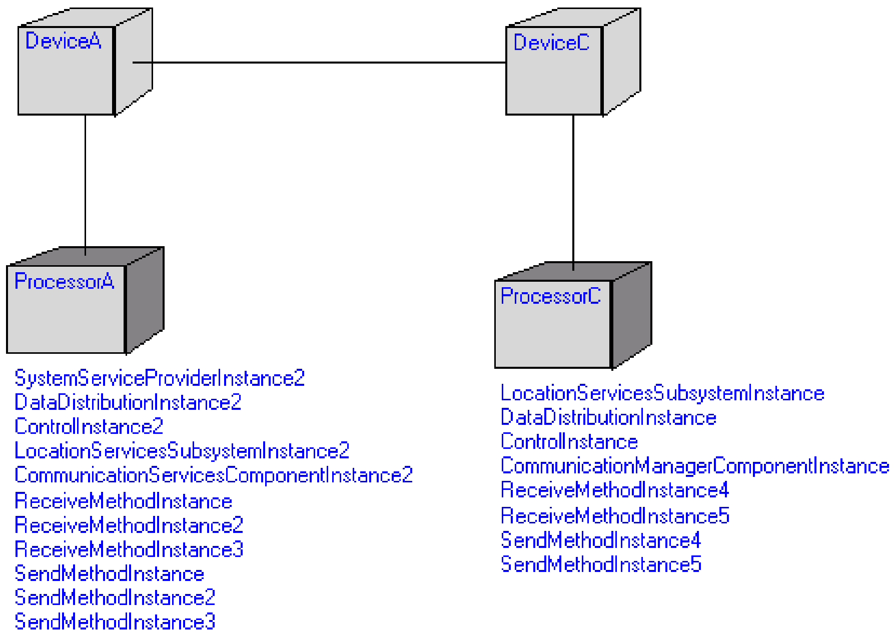


Figure 44. Concrete deployment view for the system with two devices.

DiseP executed in ProcessorA is in the role of a system service provider by executing system services (SystemServiceProviderComponentInstance). It also has three different communication methods (i.e. receive method – send method pairs) to communicate with a platform allocated to ProcessorC.

9.6 Analyzing the concrete architecture

The first model of the concrete product line architecture in the domain has been designed. Figure 45 describes the fine-grain external context of the distributed services platform identifying the main external actors. This fine-grain context is important in analysis as much as the other architecture views because it reveals other new external actors that can interact with a new possible product member.

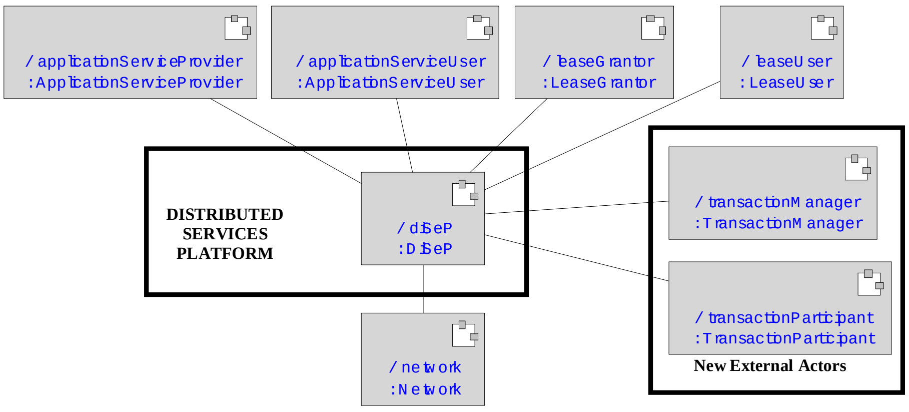


Figure 45. External actors of the product-line architecture.

In the structural view shown in Figure 46 the abstract components of the architecture are included. All the architectural elements are subsystems that are common to all product members. The system services provided, contained in the *SystemServiceProvider* subsystem, are activated by a *Control* subsystem. The system services provided communicate with the other subsystems, *DataDistribution* subsystem, *LocationServices* subsystem and *Communication-Services* subsystem.

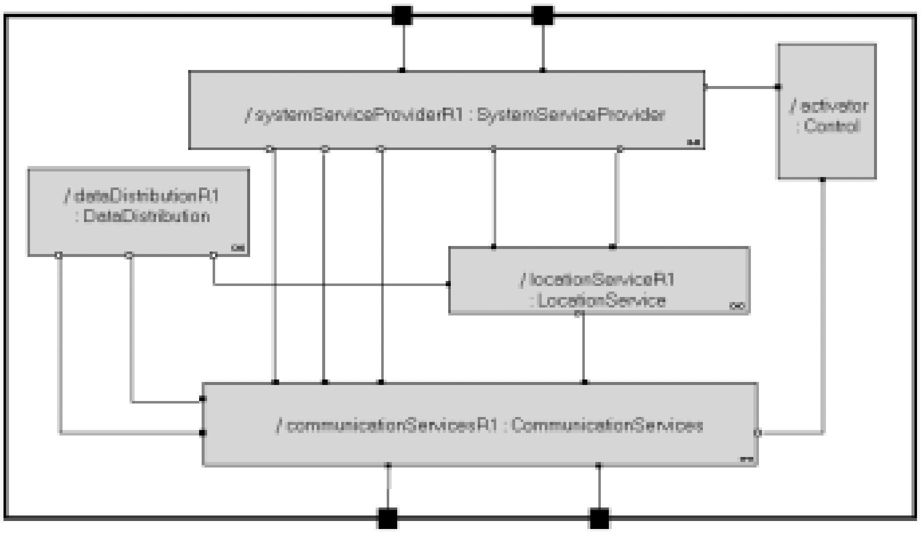

/ systemServiceProviderR1 : SystemServiceProvider
/ activator : Control
/ dataDistributionR1 : DataDistribution
/ locationServiceR1 : LocationService
/ communicationServicesR1 : CommunicationServices

Figure 46. Structural view of architecture subsystems.

For the moment, in the architecture two variability points are included (see Figure 39), but others are implicit or unspecified. The upper level services inside the *SystemServiceProvider* represent one of the variable points. In one of the product members there are two services: Lease Service and Directory Service that always come together. In other (future) products there might also be Security Service, Transaction Service, etc.

9.6.1 Adaptability analysis

We define the adaptability as the flexibility of a software product to incorporate changes in the technical requirements. The scenarios simulate the use of the distributed service platform architecture in situations with diverse technical requirements. The architecture is usable in a situation when the scenario has an impact of level three or lower. The results of these scenarios are summarized in Table 18.

Scenario 1. Which changes are needed when the architecture is to be used for secure systems?

We assume that for secure systems a number of things are necessary. First, each service user/provider/grantor action should be authenticated and it should be possible to grant different levels of access to users (no access, read-only, full control, etc.). This is already supported by the distributed service platform architecture, so it is unaffected. Second, the communication between components should be encrypted. Encrypted communication is not yet present in the architecture, but it could be added by changing one component – communication service. Finally, access to services should be prohibited for unsecured units. This means that the location service manager should be changed so that it inspects the network address of clients. The conclusion is that using the architecture for secure systems necessitates changes in a number of existing components and, therefore, this scenario has a level three impact.

Scenario 2. Which changes are needed when the architecture is to be used for real-time systems?

The distinguishing features of real-time systems are deadlines and synchronization between the different parts of a system [40]. These features are partially supported by the DiSeP PLA considering the variability included in the *CommunicationServices* subsystem represented by an optional *asynchronous messaging* component. This component is not needed in the current product where communication is synchronous only. The asynchronous messaging component could be left out for the current product and added if the architecture is to be used in real-time systems.

However, deadlines could be enforced by introducing something like a deadline manager into the control layer which makes sure that a system responds within a certain period. Similarly, synchronization could be added by introducing a synchronization manager that makes sure that the different parts of the distributed platform operate in harmony. In addition, the division of the platform into layers is perhaps not usable for real-time systems. So, the impact of this scenario is architectural and it is classified as level four.

Scenario 3. Which changes are needed when the architecture is to be used for ultra-reliable systems?

In ultra-reliable systems both software and hardware are often replicated [43]. This redundancy makes sure that the system remains in working order after one or more services have failed. This is a significant challenge because there are many different types of faults that can occur in a distributed platform.

PLA reliability is taken into account from the redundancy point of view: Thus, all data in the system has redundancy as a parameter. Data distribution manager fulfills this requested redundancy by negotiating with other data distribution managers in the network and saving replicas of data into other data storage. Also all the service proxies and other service related information is replicated. If system services functionality fails for some reason, it is still possible to recover by finding the important data from the network and activating the system services in some other unit.

In addition, these systems could use voting, which means that the same operation is performed by two or more elements, and the end result of the operation is some kind of weighted average of the results of individual elements. Both redundancy and voting could be addressed by modifying one or more components that encapsulate the access to the other services. This needs modification of more than one component and, therefore, the impact of this scenario is classified as level three.

Scenario 4. Which changes are needed when another type of interface is considered for a service user/provider/grantor?

The user of DiSeP is an application that uses this platform to deliver application services to other applications. User interface components consist of interface method calls.

To make the DiSeP services accessible from another type of interface, the platform user interface components should be adapted. Because the lower layers are independent of the platform user interface, they are unaffected by changes in the platform user interface layer. The components of this layer are the only components affected and thus the impact of this scenario is classified as level three.

Scenario 5. Which changes are needed when the architecture is used for a system that uses workflow management?

The distributed service platform already has a control service that controls which services may be performed (activate/deactivate) in a certain situation. This component could be enhanced to support true workflow management. Since the control service is the only component affected, the impact of this scenario is level two.

Scenario 6. Which changes are needed when the architecture is used for a system that uses mobile computing?

Mobility is a requirement of PLA. For the moment system services are just activated and deactivated, not actually moved. The data contents of these services are distributed in the network as replicas. The distributed service platform already has a directory service as a mobility-enabled naming service that gives global visibility to all authorized clients.

In order to support mobile computing a mobility services layer should be added above the system services. This layer has to contain a virtual resource management service component and other component services for the user, virtual environment and mobile virtual terminal [11]. Since a new layer should be added, the impact of this scenario is level four.

As expected, we see in Table 18 that the architecture is not directly usable in every situation. Using it for real-time or ultra-reliable systems necessitates major changes to the architecture. In the other situations, the architecture is usable, but some changes are necessary. When the distributed services platform architecture is used in an actual situation, more scenarios are probably needed to evaluate whether the right services are identified to encapsulate the expected changes to the technical requirements. In Table 18 the following legend has been used: – = unaffected, + = needs to changed, O = one component affected, M = more components affected.

Table 18. Summary of the scenarios for technical requirements adaptability.

Scenario	**Distributed services platform**		**Impact level**
	Architecture	**Components**	
1	–	M	3
2	+	M	4
3	+	M	3
4	–	M	3
5	–	O	2
6	+	M	4

The revolutionary approach to initiate a PL is a highly iterative process. This analysis drives us to the conclusion that first a concrete disserted functionality should be designed and then pay attention to this quality attribute.

9.6.2 Portability analysis

Portability is another important quality requirement that is placed in the technical environment category. At first sight, portability and adaptability very much look alike, but they are not the same. Portability is the ease with which a system can be adapted to changes in the *technical environment* and adaptability is the ease with which a system can be adapted to changes in the *technical requirements*. The scenarios in this category explore the effect of changes in the technical environment.

Scenario 1. Which changes are needed when another end point device is used?

The architecture contains both optional and passive components, so the architecture requires no changes.

Scenario 2. Which changes are needed when another network protocol is used?

Another network protocol would require changes in the CommunicationService component.

In Table 19, we observe that changes in the technical environment affect very few components of the DiSeP architecture. We notice that the platform actually encapsulates access to the environment. However, there may be potential changes in the technical environment, not mentioned here, that have an impact above level two. In Table 19 the legend is as follows: – = unaffected, + = needs to changed, O = one component affected, M = more components affected.

Table 19. Summary of the scenarios for portability.

Scenario	**DiSeP architecture**		**Imp. Level**
	Architecture	**Components**	
1	–	–	1
2	–	O	2

10. Experiences of the use of the QADA method

This section discusses the experiences of the use of the QADA method. Experiences in design and analysis i.e. in Quality-driven Architecture Design (QAD) and Architecture Quality Analysis (AQA) are concerned individually.

Tool support is closely related to any method, but it is obvious that no commercial tool alone supports the quality architecture design method [8]. Instead, two different tools were used when applying the QADA method in practice: RoseRT [51] and Visio [45]. In addition to these tools, a text editor with text processing effects (tabs, bolding, italic etc.) and the possibility to create and edit tables is needed. The appendix A describes the configuration and use of the tools.

10.1 The QAD method

We started by studying the ability of existing design methods to describe the characteristics of software architecture. The three methods, with four, 4+1 and 3+1 views, have differences in the naming of views, their descriptions and notations. However, none of these methods concerns the product line architecture development. In most cases, UML is used as a modeling language but also tables, natural languages and other modeling languages are appropriated. Therefore, we drew the conclusion that simplicity, consistence and understandability are the key issues of most importance in a method applicable for the development of product line 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 utilizes architectural styles and patterns as a guide to carry out quality requirements in architectural descriptions.

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 modeling 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 we could not have anticipated in this early phase of the method development.

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.

In addition to these, QAD should support the linking of the architectural views with the development process and assets management. This support could be implemented by adding a separate development view to the QAD method.

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.

10.2 The AQA method

Considering the quality analysis at the conceptual level, it is important to examine the relationship between architectural views and architectural styles, as the architectural style is also considered to have an impact on the quality attributes of the system. However, there is little research that examines this relationship. In this way, the result of the examination has to respond to questions such as: (1) which architectural view the architectural style is focused on, (2) what specific quality attributes the style is considered to support, and (3) what kind of assumptions are made about context or environment.

The quality analysis of the concrete architecture permits obtaining better results that could improve the design.

This report represents the first step toward providing guidelines in the domain of middleware services. PLA of the distributed service platform is in the first form of a software development cycle and we tried to model it by the means of applying a revolutionary approach for PL initiation. The development of the distributed servicesplatform PLA is an iterative process, as is the analysis. One of the goals of the analysis of an early design of PLA is relevant for the product-line features uncovered. In the concrete architecture we analyzed adaptability and portability, as development quality attributes using a scenario-based method similar to SAAM. We consider that is very important to consider economic aspects in the analysis. We couldn’t apply a method based on customer value analysis, due to the novelty of the domain concept on the market and the lack of an economic data model. Also we couldn’t identify related studies in the literature for the product-line architecture analysis methods in the domain of middleware services.

11. Conclusions

In this report we introduced and demonstrated a systematic method where quality analysis is associated with the quality design in an iterative improvement of the product line architecture when the software product-line is initiated in a revolutionary style. Our practical experience shows that the proposed method promotes design and analysis at multiple resolutions as a means of minimizing risk with acceptable levels of time and effort.

Thus, considering the design issues, three viewpoints of software 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 in a systematic way, tool support is needed and hence the method experimented with existing tools for a distributed service architecture case study.

In order to fulfill the quality requirements that have been set for a software product-line, a strong consideration of architectural viewpoints in the software development is required. Also, in order to reach quality attributes with architectural structures, the use of architectural styles and patterns is needed.

In addition to quality requirements, software architecture has to answer to the functional demands defined by the customers and end-users. Placing heavy 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 the concrete structures and activities is needed.

The quality analysis method for both conceptual and concrete architecture descriptions applies only questioning techniques. The product line architecture must not only conform to the quality requirements of each product line member, but it must also be generic and adaptable to the whole product line domain. It is important to know how reusable and flexible to anticipated changes the product-line architecture is, such as to maximize the reusability and to minimize possible changes in the functionality required by various product members.

Assuming that the benefits and drawbacks of each architecture style in relation to quality attributes are already known, the analysis of conceptual descriptions has the objective to check architecture styles and violations of the standard patterns. Also, the role of the architecture analysis at the conceptual level is to provide a knowledge base for the domain architecture so as to perform a more comprehensive analysis of quality attributes at the concrete level description. Thus, the experts' knowledge can be better structured and used in a more systematic way to generate scenarios associated with the most important quality attribute of the domain.

Concrete architecture descriptions permit more relevant and accurate scenario-based analysis method results for the development of non-functional quality attributes such as reusability and adaptability.

Utilizing the architectural constructs mentioned during our report, the QADA method provides an explicit and quality-driven link between software requirements and architecture. The open problem of a product line architecture design and analysis method is how to take better advantage of architectural concepts to analyze the software product line for quality attributes in a systematic way. It is also very important to identify potential risks and to verify that the quality requirements of the PL domain have been addressed in the PLA design.

In future research we want to further validate our QADA method, whether the identified views at the conceptual and concrete architecture levels capture all relevant information needed for the quality-driven software product-line development. Other additional views are needed and the extent to which the set of needed views depends on the product-line domain concerned or the list of quality attributes analyzed.

In addition, our aim is to refine the semantics of the views used and to define the requirements of the extensions needed in standard UML from each viewpoint of the QADA method.

Acknowledgements

We would like to thank Prof. Anne Eerola, University of Kuopio, and Prof. Veikko Seppänen, University of Oulu, for providing encouraging critique and guiding comments as the reviewers of this report.

References

1. Abowd, G., Bass, L., Clements, P., Kazman, R., Northop, L. and Zaremski, A. Recommended Best Industrial Practice for Software Architecture Evaluation. CMU/SEI-96-TR-025. 1997.

2. Anthony, R. J. A Taxonomy of Transparency and a Dependency Graph. Proceedings of IASTED'01, ACTA Press. 2001. Pp. 692 - 699.

3. Bachmann, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R. and Little, R. Software Architecture Documentation in Practice: Documenting Architectural Layers. Special report, CMU/SEI. 2000.

4. Bachmann, F. and Bass, L. Managing Variability in Software Architectures. Proceedings of SSR’01, ACM. 2001. Pp. 126 - 132.

5. Bachmann, F., Bass, L., Chastek, G., Donohoe, P. and Peruzzi, F. The Architecture Based Design Method. Technical report, CMU/SEI. 2000.

6. Barbacci, M., Ellison, R., Weinstock, C. and Wood, W. Quality Attribute Workshop Participants Handbook. CMU/SEI-2000-SR-001. 2000. 44 p.

7. Barbacci, M., Klein, M. and Weinstock, C. Principles for Evaluating the Quality Attributes of a Software Architecture. Technical Report, CMU/SEI-96-TR-036, ESC-TR-96-136, 1997.

8. Bass, L., Clements, P., Donohoe, P., McGregor, J. and Northrop, L. Fourth Product Line Practice Workshop Report. Technical report, CMU/SEI. 2000.

9. Bass, L., Clement, P. and Kazman, R. Software Architecture in Practice. Addison-Wesley. 1998.

10. Bass, L., Klein, M. and Bachmann, F. Quality Attribute Design Primitives and the Attribute Driven Design Method. Proceedings of PFE-4. 2001. Pp. 163 - 176.

11. Bellavista, P., Corradi, A. and Stefanelli, C. Mobile Agent Middleware for Mobile Computing. IEEE Computers, March 2001, pp. 73 - 80.

12. Bosch, J. Design & Use of Software Architectures. Addison-Wesley. 2000.

13. Bratthall, L. and Runeson, P. A Taxonomy of Orthogonal Properties of Software Architectures. Proceedings of NOSA'99, University of Karlskrona. 1998.

14. Bray, T., Paoli, J., Sperberg-McQueen, C. M. and Maler, E. Extensible Markup Language (XML) 1.0. 2. Ed. 2000.

15. Buschmann, F., Meunier, R. and Rohnert, H. Pattern-oriented software architecture, a system of patterns. John Wiley & Sons. 1996.

16. Coplien, J., Hoffman, D. and Weiss, D. Commonality and Variability in Software Engineering. IEEE Software, Nov/Dec pp. 37 - 45. 1998.

17. Coriant, M., Jourdon, J. and Boisbourdin, F. The SPLIT Method. SPLC Conference, Kluwer Academic, pp. 147 - 167. 2000.

18. Dobrica, L. and Niemelä, E. A Strategy for Analysing Product Line Software Architectures. VTT Publications 427, Espoo, Technical Research Center of Finland. 2000. 124 p.

19. Dobrica, L. and Niemelä, E. Attribute-based product-line architecture development for embedded systems. Proceedings of AWSA'00, Monast University, 2000. Pp. 76–88.

20. Dobrica, L. and Niemelä, E. A Survey on Software Architecture Analysis Methods. Accepted to IEEE Trans on Soft. Eng. 2001.

21. Faulk, S., Harmon, R. and Raffo, D. Value-Based Software Engineering (VBSE), A Value-Driven Approach to Product-Line Engineering. Proceeding of SPLC1. Kluwer Academic Publishers. 2000.

22. Fayad, M. E., Schmidt, D.C. and Johnson, R. E. Building application framework: Object-Oriented Foundation of Framework Design. Wiley Computer Publishing, 1999.

23. Gamma, E. Design patterns: elements of reusable object-oriented software. Addison-Wesley. 1994.

24. Gomaa, H. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley. 2000.

25. Hauser, J.R. and Clausing, D. The House of Quality. Harvard Business Review. May-June, pp. 63 - 73. 1988.

26. Hein, A., Schlick, M. and Vinga-Martins, R. Applying Feature Models in Industrial settings. SPLC, Kluwer Academic Publisher, 2000, pp. 47 - 71.

27. Hofmeister, C., Nord, R. and Soni, D. Applied Software Architecture. Addison-Wesley. 2000.

28. IEEE Computer Society, IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems. IEEE Std-1471-2000.

29. IEEE Standard 1061-1992. Standard for software quality metrics methodology. New York: Institute of Electrical and Electronics Engineers, 1992.

30. IEEE Standard Glossary of Software Engineering Terminology. IEEE Std. 610.12-1990.

31. ISO/IEC91 – International Organization of Standardisation and International Electrotechnical Commission. *Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use*. ISO/IEC 9216: 1991(E).

32. Jaaksi, A., Aalto, J-M., Aalto, A. and Vättö, K. Tried & True Object Development. Industry-Proven Approaches with UML. Cambridge University Press. 1999.

33. Jacobson, I. Object-oriented software engineering: a use case driven approach. Harlow: Addison-Wesley. 1992.

34. Jacobson, I., Grissand, M. and Jonsson, P. Software Reuse: architecture, process and organization for business success. Addison Wesley. 1997.

35. 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. Pp. 143 - 168.

36. Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T. and Carrière, S. J. The Architecture Tradeoff Analysis Method. Proceedings of ICECCS, Monterey, CA, August 1998. Pp. 68 - 78.

37. Kronlöf, K. Method integration: concepts and case studies. John Wiley & Sons. 1993.

38. Krutchen, P. B. The 4+1 View Model of Architecture. IEEE Software, Vol. 12, Nov 1995. Pp. 42 - 50.

39. Lalanda, P. Product-line software architecture. PRAISE project deliverable number 2.2. http://www.esi.es/Projects/reuse/Praise/March 1999.

40. Laplante, P. Real-time systems design and analysis. IEEE Press, New York. 1993.

41. Lassing, N. H., Rijsenbrij, D. B. B. and Vliet, J. C. van. Flexibility in ComBAD architecture. In proceedings of First Working IFIP Conf. on Software Architecture (WICSA1), San Antonio, Texas, February 1999.

42. Leishman, D. A. Solution Customization. IBM Systems Journal, Vol. 38 (1), 1999. Pp.76 – 97.

43. Leveson, N. Safeware: systems safety and computers. Addison Wesley, Reading, 1995.

44. Meekel, J., Horton, T. B. and Mellone, C. Architecting for domain variability. Second International Workshop on Development and Evolution of Software Architectures for product line, LNCS 1429, Springer Verlag, 1998.

45. Microsoft Visio 2000, visual business language tool. http://www.microsoft.com/office/visio/

46. Niemelä, E. A component framework of a distributed control systems family. VTT Publications 402, Espoo, Technical Research Center of Finland. 1999.

47. Niemelä, E. and Ihme, T. Product line software engineering of embedded systems. Procs of SSR'01, Symposium on Software Reusability. Toronto, Ontario, CA, 18 - 20 May 2001 (2001), pp. 118 - 125.

48. Parnas, D. On the design and development of product families. IEEE Transactions on Software Engineering, SE-2 (1), 1976.

49. Perry, D. and Wolf, A. Foundation for the Study of Software Architecture. SIGSOFT Software Engineering notes, Vol. 17, No. 4, 1992, Pp. 40 - 52.

50. Pree, W. Design patterns for object-oriented software development. Addison Wesley/ ACM Press, 1995.

51. Rational Rose RealTime CASE tool. http://www.rational.com/products/rosert/index.jsp.

52. Rational Rose RealTime Modeling Language Guide. Rational Software Corporation, Version 2000.02.10.

53. Ramaswamy, R. and Nerurkar, U. Creating malleable architectures for application software product families. Workshop on object technology for product line architectures (OPLA 99), Lisbon, 14 - 18 June 1999.

54. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. Object-oriented modeling and design. Prentice Hall. 1991.

55. Rumbaugh, J., Jacobson, I. and Booch, G. The Unified modeling language reference manual. Addison-Wesley. 1999.

56. Selic, B., Gullekson, G. and Ward, P. Real-time object oriented modeling. John Wiley & Sons. 1994.

57. Selic, B. and Rumbaugh, J. Using UML for Modeling Complex Real-Time Systems. Rational Software White paper, 1998. http://www.rational.com/products/whitepapers/100230.jsp.

58. Shaw, M. and Garlan, D. Software Architecture. Perspectives on an Emerging Descipline. Prentice Hall. 1996

59. Stevens, W., Myers, G. and Constantine, L. Structured design. IBM Systems Journal. Vol. 2, No. 13. 1974. Pp. 115 - 139.

60. Szyperski, C. Component Software. Beyond Object-Oriented Programming. New York: Addison Wesley Longman Ltd. 1997.

61. TINA Consortium Service Architecture specification. http://www.tinac.org

62. Unified Modeling Language 2.0 Superstructure, Request For Proposal. http://cgi.omg.org/cgi-bin/doc?ad/00-09-02.

Appendix A: Tools used in the QADA method

In this appendix we discuss how the tools were used in applying the QADA method in practice. The tools were a *text processing tool*, and *conceptual* and *concrete architecture design tools*.

Text processing tool

A text editor is needed in the early phase to enumerate and cluster the responsibilities of a system. Experiences in using a simple textual list as a means of recording system responsibilities were encouraging. This working method seems to be useful and simple not only for creating, but especially in updating and managing responsibilities. Because the list of responsibilities is used along with the structural model to illustrate the main functions of the static components, it is essential that a list contain only up-to-date information. If updating an architectural description is this easy, there is a low threshold to do it. Moreover, a list of quality attributes assists architecture analysis by identifying meaningful targets and defining the scope for the analysis.

Comparing the initial list in Figure 1 with the completed list illustrated in three parts in Figure 33, Figure 32 and Figure 34 shows the evolution of the responsibilities during the design process. In Figure 1, the system responsibilities are more inaccurate and decomposed in a different way than in the final version. This early clustering of functionality really has an affect on the final architecture. The evolution of lists included at least five different versions between the draft and the completed version.

- Use system services:
  - Locate a system service provider
    - Listen for beacon signals and keep track of the system service providers through them
  - Distribute data over distributed data storage:
    - Allow application to use application services through a directory service interface:
      - Allow application to search for suitable services
      - Allow application to fetch a service proxy
    - Allow application to provide application services through a directory service interface:
      - Allow application to create an appropriate service proxy
        - Create a unique service number for the proxy
      - Allow application to register the service proxy to a directory service
      - Allow application to unregister the service proxy from the directory service
  - Utilize lease management between two independent elements:
    - Take care of lease renewals for any leased system resource
    - Keep track of lease renewals for any shared and leased resouce
  - Perform and track transactions to reach synchronized operations between elements:
    - Prepare, start and abort transactions
- Provide system services: (System service units)
  - Announce the availability of the system services
    - Send beacon signals about the location and the operational state of the system service provider
  - Contribute to the operation of a distributed data storage
    - Allow data to be stored in local resources
    - Track that the need for redundant data copies is satisfied
    - Negotiate about the copying, transferring or deleting the data if necessary
  - Provide a directory service interface to the distributed data storage
    - Register, unregister, search and fetch for services
  - Contribute to producing a lease service interface
    - Accept and host leases of lease grantors
    - Grant leases of lease grantors for lease users
  - Provide a transaction service interface
- Monitor the state of the network:

Figure 1. Functional list of responsibilities, the first draft.

Conceptual architecture design tool

The visual business language tool Visio, and especially its UML stencils, is used at the abstract level to document the conceptual structure (an example in Figure 11) and behavior (an example in Figure 12). The conceptual deployment model (an example available in Figure 13) is drawn up with basic Visio drawing shapes. Visio provides a flexible means to use UML elements with different basic shapes, which is an important feature when making abstract drafts of software architecture.

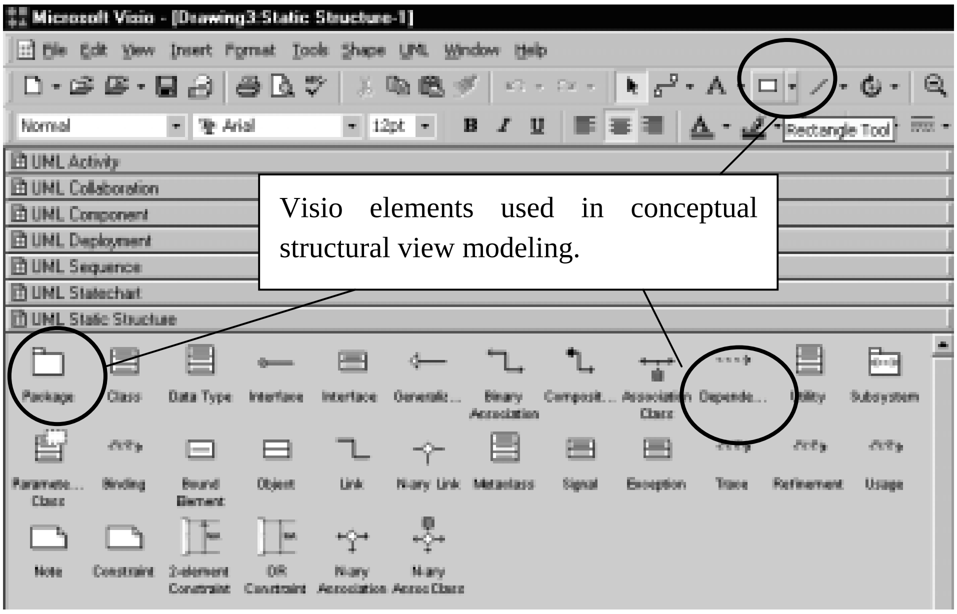


Figure 2. UML stencil for static structure modeling in Visio2000.

Figure 2 illustrates the user interface of the Visio and also shows, which modeling and drawing elements are used in the modeling of the conceptual structural view. Similarly the UML collaboration stencil is used to model the conceptual behavior by the means of collaboration scenarios. Collaboration stencil of Visio is configured to meet the requirements of conceptual behavior modeling by adding the “Actor” modeling element from the use case stencil. All the needed modeling elements needed are shown in Figure 3.

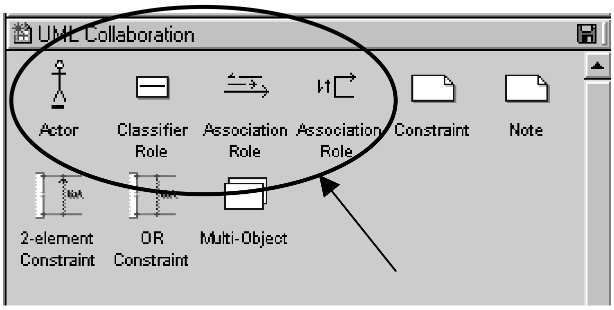


Figure 3. UML stencil for behavior modeling in Visio2000.

Design rationale for modeling elements is recorded in a separate "Documentation" tab, which is located in the View->Status bar window.

Concrete architecture design tool

The concrete architecture is modeled with the RoseRT CASE tool. It supports structure diagrams with capsules and ports and also state, sequence and deployment diagrams (example models available in Figures 15–18). Capsule stereotypes help to illustrate the decomposition level of components.

Configuration of the RoseRT

Modeling the concrete architecture with Rose RealTime is more fluent after the few configuration steps described below.

1. Model framework selection

The model framework selection is done according to the desired programming language of the executable model. The model framework is selected every time a new model is created. The supported frameworks are:

- Empty
- RTC
- RTC++ and
- RTJava

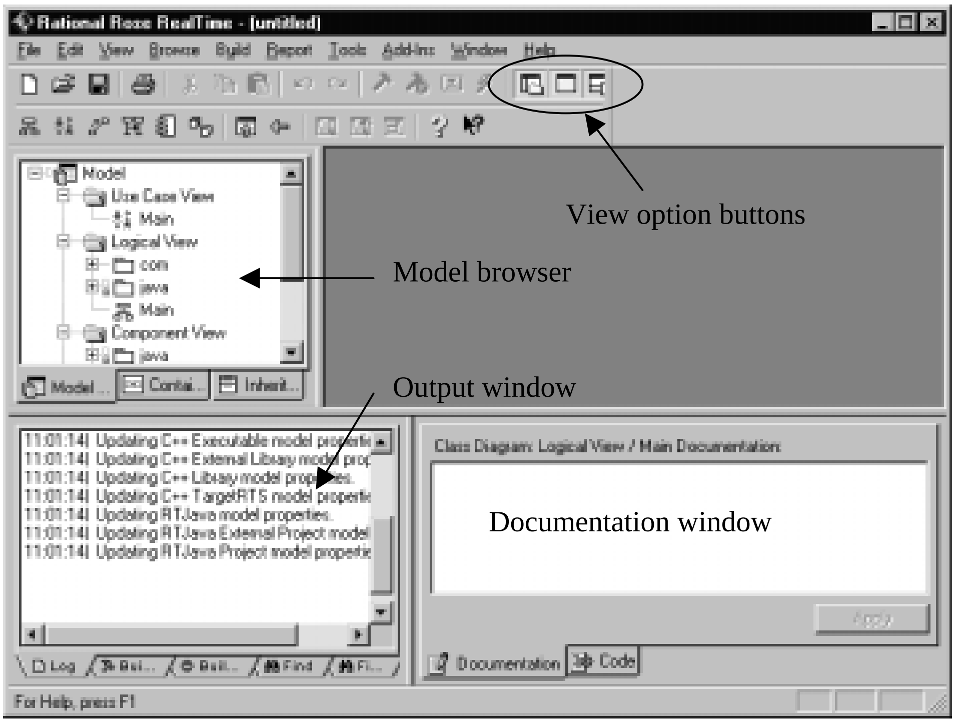
In this case study the RTJava framework was used, which means that the code generated for the executable model is in the Java programming language.

The second configuration step after model framework selection is the tool user interface configuration, which is done through *selecting* the view options, *customizing* the diagram toolbars and *filtering* the model browser.

2. Selecting the view options

The view options are selected to be as in Figure 4, from left to right

- View Browser(s), yes, makes Model view browser visible
- View Description, yes, shows Documentation on the bottom of the window
- View Output, yes/no, required only if generating executable models


Rational Rose RealTime - [untitled]
File Edit View Browse Build Report Tools Add-Ins Window Help
View option buttons
Model browser
Model
Use Case View
Main
Logical View
com
java
Main
Component View
java
Model... Contai... Inherit...
Output window
Class Diagram: Logical View / Main Documentation:
Documentation window
Apply
Log Bui... Buil... Find Fi...
Documentation Code
For Help, press F1

```
11:01:14| Updating C++ Executable model properti
11:01:14| Updating C++ External Libray model prop
11:01:14| Updating C++ Libray model properties.
11:01:14| Updating C++ TargetRTS model properti
11:01:14| Updating RTJava model properties.
11:01:14| Updating RTJava External Project model
11:01:14| Updating RTJava Project model properti
```

Figure 4. RoseRT user interface.

3. Customizing the toolbars

After selecting the view options, the diagram toolbars are customized. Clicking the toolbar with the right button of mouse activates the customizing window. In order to customize the diagram toolbar you have to have a diagram in question open and active. Figure 5 shows, which tools are required in the structural view design of the concrete architecture.

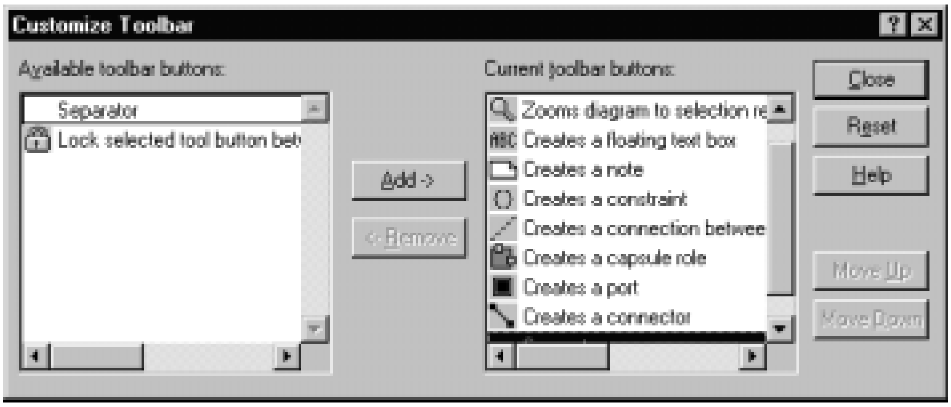


Figure 5. Customizing the structure diagram toolbar.

In addition to the structure diagram toolbar, the state diagrams and the message sequence chart toolbars have also to be customized. This is shown in Figure 6, Figure 7 and Figure 8.

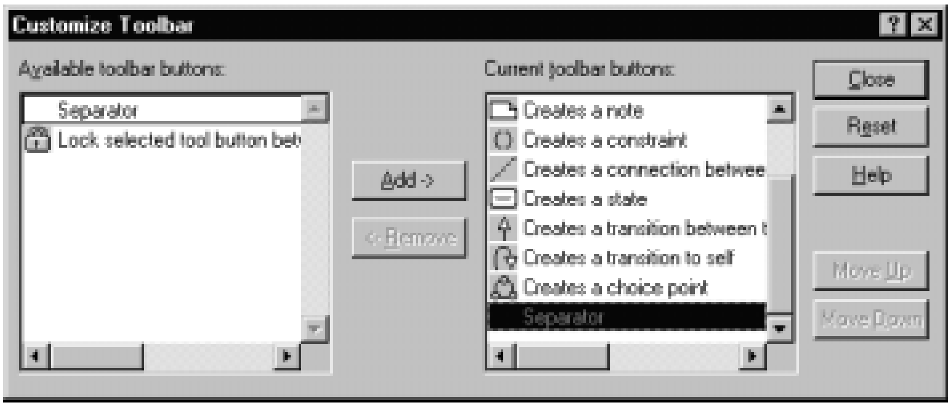


Figure 6. Customizing the state diagram toolbar.

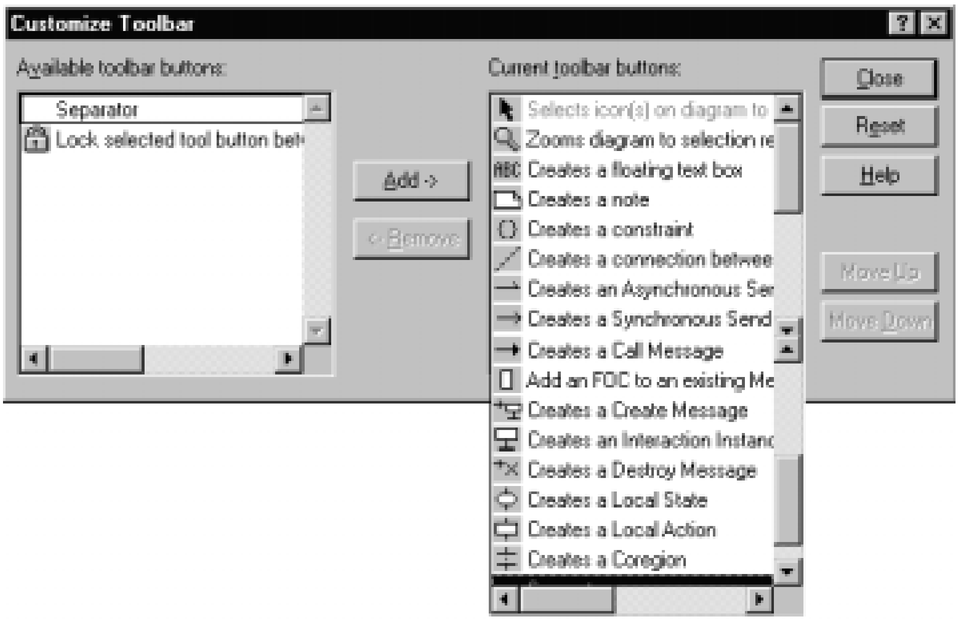

Customize Toolbar
Available toolbar buttons:
Separator
Lock selected tool button bet
Add ->
<- Remove
Current toolbar buttons:
Selects icon(s) on diagram to
Zooms diagram to selection re
Creates a floating text box
Creates a note
Creates a constraint
Creates a connection betwee
Creates an Asynchronous Ser
Creates a Synchronous Send
Creates a Call Message
Add an FOC to an existing Me
Creates a Create Message
Creates an Interaction Instanc
Creates a Destroy Message
Creates a Local State
Creates a Local Action
Creates a Coregion
Close
Reset
Help
Move Up
Move Down

Figure 7. Customizing the MSC toolbar.

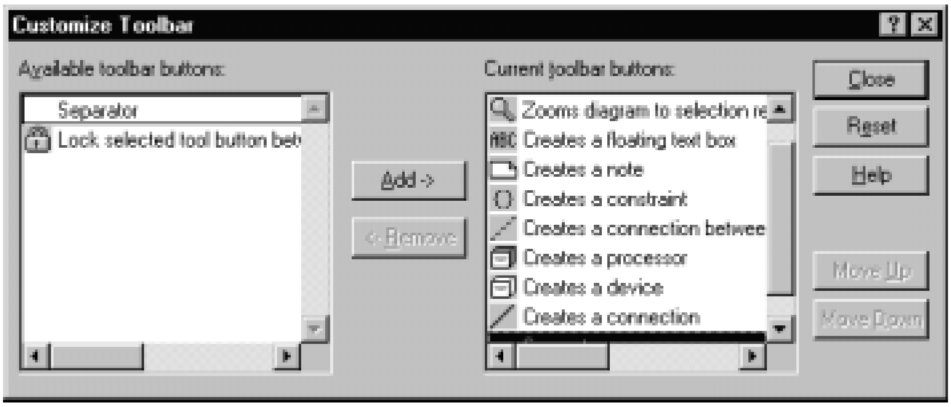
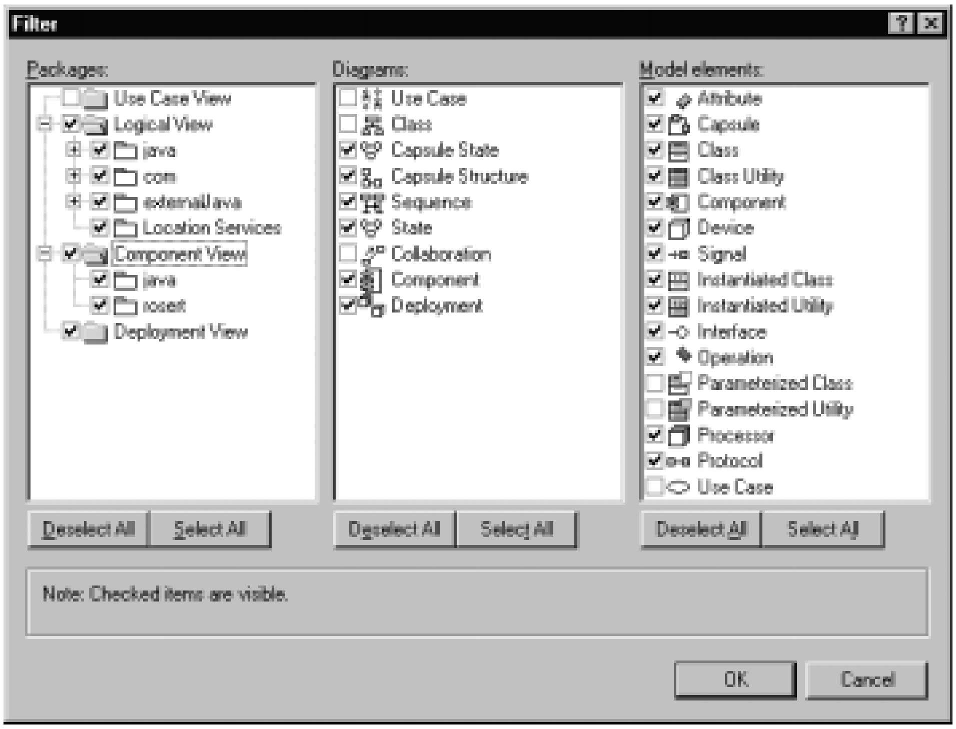


Figure 8. Customizing the deployment diagram toolbar.

4. Filtering the Model Browser

The Model Browser is filtered by deselecting packages, diagrams and model elements that are not used in your model. In this case it mainly means the Use Case View and diagrams and elements related to it (Figure 9).



Filter

Packages:	Diagrams:	Model elements:
☐ Use Case View	☐ Use Case	✓ Attribute
✓ Logical View	☐ Class	✓ Capsule
✓ java	✓ Capsule State	✓ Class
✓ com	✓ Capsule Structure	✓ Class Utility
✓ externalJava	✓ Sequence	✓ Component
✓ Location Services	✓ State	✓ Device
✓ Component View	☐ Collaboration	✓ Signal
✓ java	✓ Component	✓ Instantiated Class
✓ rosert	✓ Deployment	✓ Instantiated Utility
✓ Deployment View		✓ Interface
		✓ Operation
		☐ Parameterized Class
		☐ Parameterized Utility
		✓ Processor
		✓ Protocol
		☐ Use Case

Note: Checked items are visible.

Figure 9. Filtering model browser.

After configuration of the workspace, it is saved and the model can be opened *with workspace.* This avoids reconfiguring the tool workspace every time.

Advantages and disadvantages

The version of the Rose RealTime case tool used was 6.3.112.0. The tool has several advantages as an architectural design tool, but also improvements are required in a few of these features. These pros and contrast for the tool are next discussed in more detail. "+" means for an advantage and "–" for a disadvantage.

Documentation

+ The documentation window enables the user to add comments directly to every individual modeling element i.e. capsule, port, protocol.

+ It is possible to generate a Word-document including component names and attached comments as a documentation of the system. There is also a possibility to add the attributes, operations and their descriptions.

– The possibility to "*generate document for selected parts of the model*" does not work properly. After selection, the generated document is discovered to be empty.

– The option to edit the order of the elements introduced in the document is disabled. This leads the generated document always to contain some unwanted information in an unexpected order and therefore requires manual processing afterwards.

Modeling elements and diagrams

+ Structural modeling elements supporting component-based development are represented in the tool in the form of capsules and ports. In addition to structural component-based elements, the behavior modeling is supported through state machines and message sequence diagrams. The tool also supports the allocation models of software components.

– Debugging the behavior of components worked properly only after a fix patch.

Variability support

+ Support for optional and plug-in capsule roles

+ Optional and plug-in capsules can be defined as substitutable i.e. alternative properties.

– Substitutability is not shown in graphical notation.

Workload allocation

+ After the definition of the concrete structure of the top capsule, the subsystem level capsules can be distributed in the work team and processed forward individually.

– Due to the iterative nature of the architecture design, the architecture team still needs to be in close co-operation after work allocation.

Model management

+ Capsule stereotypes like <<subsystem>> or <<component>> help in managing the hierarchical containment within capsules.

– It is not possible to categorize structural and behavior models in separate folders as QAD categorizes these models in separate views. Instead, behavior is always attached to an individual component.

– Stereotypes disabled the model-debugging feature

– Instead of hierarchical listing, which obviously would be the most logical and simplest way to do it, the ordering of components in the model browser window is done in an alphabetical order.

Quality of support

+ Technical support is quick, effective and always willing to serve.

– Complete Java tutorials are not available

Maturity

+ This tool has been developed since mid 1990. First as a tool called ObjectTime and later with the name Rose RealTime for Windows.

– The first version including a Java generator was launched in December 2000.

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Publications 456
VTT–PUBS–456

Author(s)
Matinlassi, Mari, Niemelä, Eila & Dobrica, Liliana

Title

Quality-driven architecture design and quality analysis method
A revolutionary initiation approach to a product line architecture

Abstract

The role of software architecture has changed. The use of modern software technologies and practices enables turning the focus of system development to the quality aspects of software instead of functional properties. Architecture addresses the quality issues of software and, therefore, it must be developed and documented properly. In particular, there is a need for high level architectural descriptions. The top down nature of software architecture design induces this need.

In this report we introduce a quality-driven architecture design and analysis (QADA) method. Quality-driven is about utilizing architectural styles and patterns as a means of designing high-quality architectures. QADA takes a revolutionary approach to the initiation process of a new product line. That is, the development of a complete product-line architecture and a set of components before developing the first product in a new domain. QADA considers architecture on two levels of abstraction: conceptual and concrete. Design produces architectural descriptions at both abstraction levels from three viewpoints: structural, behavior and deployment. The structural viewpoint is concerned with composition of software components, whereas the behavior viewpoint takes the behavioral architecture aspects under consideration. The deployment viewpoint refers to embedding and allocation of software components to various computing environments. Quality of architecture on both levels of abstraction is analyzed in the corresponding analysis phases.

Because software architectural design is difficult to discuss merely at an abstract level, the QADA method is tested with a case study of a distributed service platform. The platform embodies a layered service architecture, thereby providing a variety of services for its users. The upper layer of services, i.e. the system services of the platform is mobile, enabling spontaneous networking.

Keywords
software product line, architecture design and analysis methods, quality attributes, service architecture, software architecture, QADA

Activity unit
VTT Electronics, Embedded Software, Kaitoväylä 1, P. O. Box 1100, FIN–90571 Oulu, Finland

ISBN			Project number
951–38–5967–3 (soft back ed.) 951–38–5968–1 (URL:http://www.inf.vtt.fi/pdf/)			E1SU00054
Date January 2002	Language English	Pages 128 p. + 10 p.	Price C
Name of project Product Line Architecture Design and Analysis Methods, PLANA		Commissioned by	
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	