

WIRELESS INTERNET SOFTWARE ENGINEERING IST-2000-30028

Title:
Architectural Guidelines

Version: 2.0
Date : 23 Oct 03
Pages : 37

Author(s):
E. Niemela, P. Lago, J. Kalaoja, A. Tikkala, M. Matinlassi, Marco Torchiano, P.Kallio, A. Taulavuori

To:
WISE CONSORTIUM

The WISE Consortium consists of:
Investnet, Motorola Technology Center Italy, Sodalía s.p.A, Sonera, Solid EMEA North, Fraunhofer IESE, Politecnico di Torino, VTT Electronics

Printed on:
15-Dec-03 10:49

Status:

- Draft
- To be reviewed
- Proposal
- Final / Released

Confidentiality:

- Public - Intended for public use
- Restricted - Intended for WISE consortium only
- Confidential - Intended for individual partner only

Deliverable ID: D4 (Part A)

Title:
Architectural Guidelines


Summary / Contents:

This document is a part of the deliverable D4 produced in the task 2.1 of the Wise project. Deliverable D4 includes four parts: Part A: Architectural guidelines, Part B: the WISA (Wireless Internet Service Architecture) architectural knowledge base and its reference architecture (WISA/RA), Part C: Analysis of the pilot architectures, and Part D: Handbook of reusable architectural assets.

This document presents a set of guidelines for describing the architecture of software systems in an abstract way, and for detailing their design in a more concrete way (for their implementation).

TABLE OF CONTENTS

1.	Overview of Architectural Documents	4
2.	Introduction	5
3.	Abbreviations	7
4.	Terminology	8
5.	Viewpoints	10
6.	Conceptual Viewpoints	14
6.1	Overview	14
6.2	Conceptual Structural Viewpoint	14
6.2.1	System context	14
6.2.2	Domain Information models	16
6.2.3	Functional structure	16
6.3	Conceptual Behavioral Viewpoint	18
6.3.1	Collaboration diagram	18
6.4	Conceptual Deployment Viewpoint	19
6.4.1	Deployment diagram	19
6.5	Conceptual Development Viewpoint	20
6.5.1	Business Model	20
6.5.2	Topology diagram	21
7.	Concrete Viewpoints	22
7.1	Overview	22
7.2	Concrete Structural Viewpoint	22
7.2.1	Essential information-oriented aspects	23
7.2.2	Essential computational-oriented aspects	23
7.3	Concrete Behavioral Viewpoint	24
7.4	Concrete Deployment Viewpoint	26
7.4.1	Deployment diagram	26
7.5	Concrete Development Viewpoint	26
7.5.1	Interfaces	26
7.5.2	Development structure	27
7.5.3	Technology layers	27
8.	Interface description model	28
8.1	The architectural level interface description	28
8.2	The transformation level interface description	29
8.3	The implementation of the interface description	30
8.4	an Example of the use of the model	32
9.	References	36


	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 3 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted

CHANGE LOG

Vers.	Date	Author	Description
1.06	Oct 7, 2003	Anne Taulavuori (VTT), Jarmo Kalaoja (VTT)	<ul style="list-style-type: none"> Added interface description model
1.05	Dec 5, 2002	Jarmo Kalaoja (VTT), Marco Torchiano (Polito)	<ul style="list-style-type: none"> Added an overview of D4 parts in intro Fixed some terminology definitions Improved language
1.04	Oct. 24 2002	Eila Niemelä	<ul style="list-style-type: none"> Final version for review
1.03	Oct. 15. 2002	Mari Matinlassi, Päivi Kallio (VTT)	<ul style="list-style-type: none"> Version for review in Kaiserslautern.
1.02	Aug. 7.2002	Jarmo Kalaoja (VTT)	<ul style="list-style-type: none"> This document (D4 Part A) now contains only guidelines for the viewpoints and their notation from document D4 v1.1, the rest of the previous document versio is moved to document D4 Part B v0.1 Removed development process related issues from this document and tried to simplify the document to be more readable Examples of conceptual notation added. Instantiated business model moved to conceptual development viewpoint and networked structure moved to conceptual structure viewpoint as system context. Examples for table formats for responsibilities and interfaces added.
1.01	July 30 2002	P. Lago (Polito)	<ul style="list-style-type: none"> Applied standard template for WISE documents Proposal for harmonized deployment and development viewpoints for conceptual and concrete architecture (see AP2 and AP3 Oulu Meeting, June 2002): see tables 3, 4, 5 and description of conceptual and concrete development viewpoints
1.00	March 06, 2002	P. Lago (Polito)	<ul style="list-style-type: none"> Inserted this history table. Inserted definitions for notation, service (detailed), application. Expected definition for service (business and technical perspectives). Inserted explanation of D4 objective and stakeholders. Inserted table and explanation about how to use viewpoints/notation in various development stages.

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification
1	Template for Pilots Architecture	T_Pilot_Architecture.doc
2	Wise Reference Architecture (Wisa)	D4 (Part B)
3	Architecture for Pilot 1	D10 (Part B)
4	Architecture for Pilot 2	D11 (Part B)
5	Architecture for Pilot 3	D6b

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 4 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted

1. OVERVIEW OF ARCHITECTURAL DOCUMENTS

The deliverable D4 contains the outcome of the work done within task 2.1 (Architecture) of WP2 – Technology.

This deliverable has been split into four distinct parts:

- Part A: Architectural Guidelines
- Part B: WISA architectural knowledge base (WISA) and Reference Architecture (WISA/RA)
- Part C: Analysis of Pilot Architectures
- Part D: Architecture Handbook

D4 Part A presents a set of guidelines for describing the architecture of software systems in an abstract way, and for detailing their design in a more concrete way (for their implementation).

D4 Part B presents part of the WISA knowledge base for wireless service engineering. It is made up of three major parts: 1) the taxonomy of wireless services, 2) the architectural style and pattern guidelines, and 3) WISE Reference Architecture (WISA/RA). These parts are used in the development of the pilot services in the 2nd and 3rd iterations of the Wise project. The reusable architectural assets are in Part D.

D4 Part C presents the results of analyses of Pilot architectures (Pilot 1 and Pilot2)

D4 Part D contains a set of reusable assets that can be used to build wireless services. This document should be read after the knowledge contained in D4B has been assimilated. The document provides three types of reusable architectural assets: 1) typical architectures that can be used as starting points to develop wireless service architectures; 2) architectural styles and patterns that can be used to develop services, and 3) a existing services that can be re-used in new services.

The documents are linked with each other and with the pilots' architectural documents, as show in Figure 1. The guidelines of D4A provide a common structure and notation for the pilots' architectures. This allows the definition of the reference architecture (D4B), the identification of patterns and typical architectures (D4D), and the analysis of the pilot architectures (D4C).

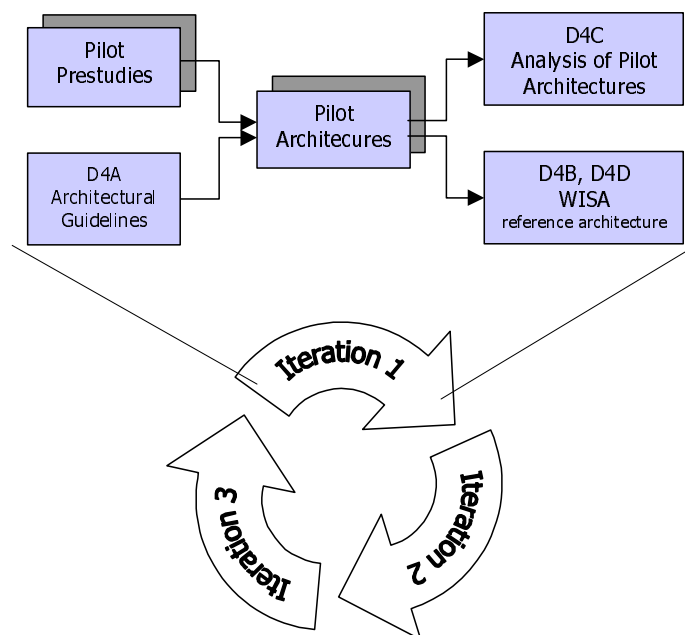



Figure 1. Organization of Architectural Documents

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 5 of 37
		Version: 1.06
		Date: 23 Oct 03
		Status : Proposal Confid: Restricted

2. INTRODUCTION

This document presents a set of guidelines that will be adopted in the WISE consortium to describe the architecture of software systems. Figure 1 sketches the complex relationships between this document, other documents and the iterative approach of the WISE project.

The main purpose of the architectural guidelines is to provide a unified and organized approach to the description of the software architecture. The architecture is described both from an abstract conceptual perspective and a detailed concrete one.

In particular, the stakeholders of this document are both *technical* and *non-technical* people. The former have to describe in detail those generic entities in terms of their implementing components. The latter have to understand the generic architectural entities making up a type of service.

The guidelines are basically made up of (1) a set of viewpoints to model the conceptual/concrete architecture (each describing a particular architectural aspect), and (2) the notation (i.e. languages and/or visual conventions) selected to model and represent each viewpoint. The views and diagrams in the architectural documents are based on these viewpoints and conform to the notation.

The contents of specific architectural documents depend on the needs emerging during the development process. Usually they contain one or more architectural views, selected among those defined by the guidelines. The table of contents of the architectural documents can be organized based on the selected views. A template for such a kind of documents is presented in "Template for Pilots Architecture" (Ref 1).

The basic concepts of software architecture with and their relationship are described in the metamodel shown in Figure 2 (adapted from [7]).

The architecture is essentially a description of a software system, from several viewpoints. Its purpose is to address the concerns of the system stakeholders.

The description consists of several views, each conforming to a viewpoint. In practice the views are made up one or more models or diagrams.

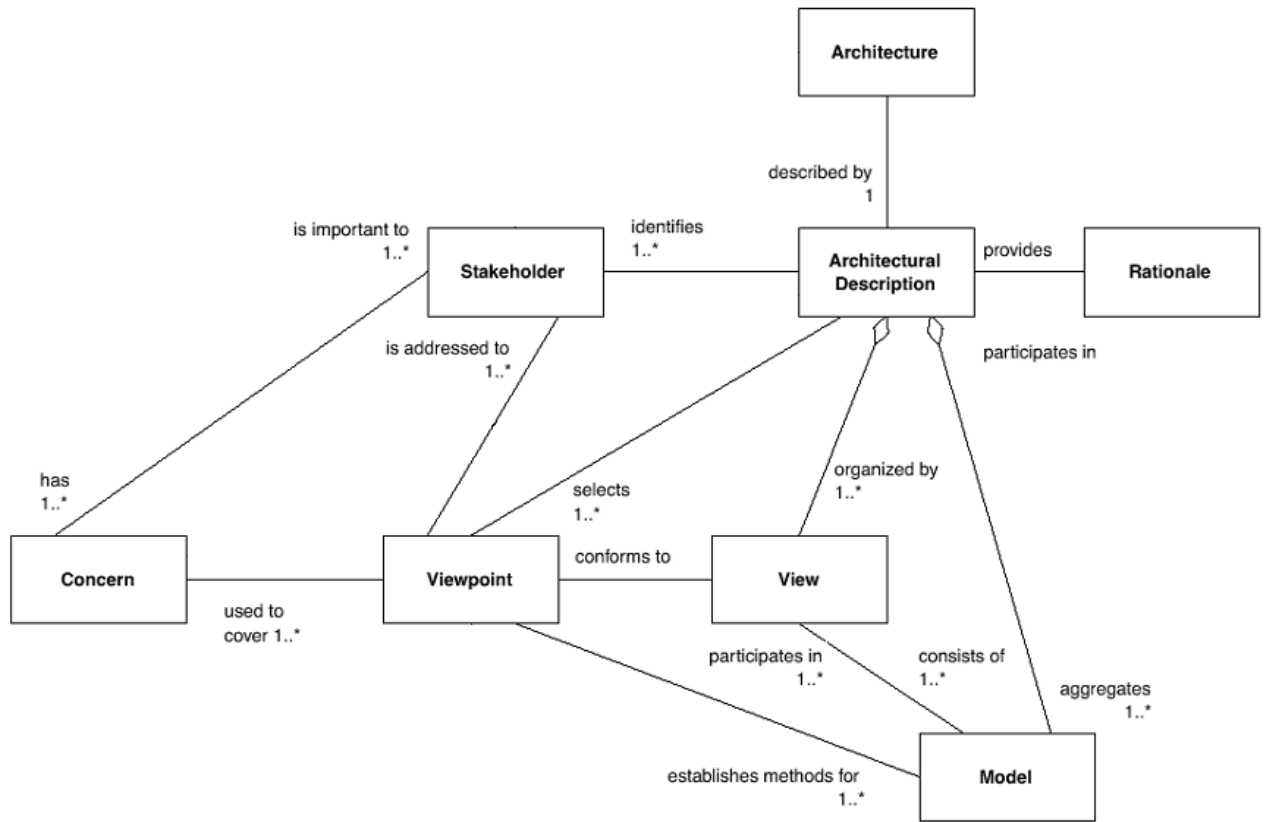
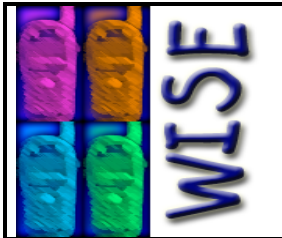




Figure 2. A metamodel of architectural descriptions (from IEEE1471)

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 7 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted


3. ABBREVIATIONS

BSS	Business Support Systems
CLDC	Connected Limited Device Configuration
CRM	Customer Relationship Management
J2ME	Java 2 Micro Edition
MIDP	Mobile Information Device Profile
OSS	Operating Support Systems
SLA	Service Level Agreement based on Monitoring and Reporting tools
SLC	Service Logic Component
TOM	Telecom Operations Management
VP	Viewpoint
COTS	Commercial Off The Shelf
MOTS	Modifiable Off The Shelf

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 8 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted

4. TERMINOLOGY

- *Software architecture* it is the structure or structures of the system, comprising the software components, the externally visible properties of those components and the relationships among them [2]. Software architecture includes also the principles and guides that control the design and evolution in time [13, 14].
- *Conceptual architecture* describes a system from an abstract level, omitting the implementation details.
- *Concrete architecture* it is detailed description of a system that addresses the implementation details.
- *Reference architecture* is a generic architecture that addresses a set of applications.
- *Specific architecture* describes the architecture of a specific system (e.g. the pilots in WISE); it can be the instantiation of a reference architecture.
- *Architectural view* is a representation of a whole system from the perspective of a related set of concerns [7].
- *Viewpoint* is a specification of the conventions for constructing and using an architectural view by establishing the purposes and audience for a view, and the techniques for its creation and analysis [7]. It can be expressed as a template used to develop specific views.
- *Architectural style* defines a class of architectures and is an abstraction for a set of architectures. 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 [2].
- *Architectural pattern* expresses fundamental structural schema for software systems, which are applied for high-level system subdivisions, distribution, interaction and adaptation [4]. An architectural pattern is strictly described and commonly available.
- *Design pattern* describes a recurring structure of communicating components, which solves a general design problem in a particular context [6]. Design patterns are micro architectures, in that they refine the subsystems or components of a software system, or the relationships between them [19]. Alone, they do not guarantee a good overall architecture.
- *Software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only [16]. This means that the component clearly specifies all dependencies with its environment, and that its interfaces realize all existing responsibilities towards potential clients. Often a component is an atomic unit of deployment.
- An *interface* defines a contract between a component requiring certain functionality and a component providing that functionality. The interface represents the main tool to specify the functionality that a component provides [3].
- A *framework* is a set of classes that embodies an abstract design for solutions to a family of related problems [8].
- *Software product family* is a set of systems belonging to the same application domain.
- *Software product line* is a group of products sharing a common, managed set of features that satisfies specific needs of a given market [2]. Software products are instances of the software product line. Each product adheres to a specific market strategy and application domain; products share architecture and are built from the components included in the product line.
- *(Software) 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 [2].
- A *software feature* represents a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems [10]. It possibly has to comply with a set of functional and quality requirements [3].
- *Mandatory feature* is a feature that must always be included to a product of a product family [10, 9].
- *Optional feature* is a feature that is contained in one or some products of a product line but not in all products [1, 10, 9, 11].
- *Alternative feature*: is one of the possible features that can fit into one of the placeholders defined by the architecture. Alternative features cannot coexist with one other in the same system [1, 10, 9, 11].

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 9 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted

- *Software platform* is considered as a means to provide shared functionality, but without any architectural constraints [3]. It is frequently referred as the top-level product-line asset set.
- *Application domain* is a set of current and future applications, which share a set of common capabilities and data [10].
- *Software reuse* is a process of implementing or updating software systems using existing software assets. Assets can be defined as software components, objects, software analysis and design models, domain architecture, database schema, code, documentation, manuals, standards, test scenarios, and plans [15].
- *Vertical reuse* is reuse of components from other systems within a domain. *Horizontal reuse* is reuse of components from other domains [15].
- *Service* is the capability of an entity (the server) to perform upon request of another entity (the client), an act that can be perceived and exploited by the client. From a business perspective, or from a technical perspective, this definition might change.
- *Service architecture* is architecture of applications and middleware. It is a set of concepts and principles for the specification, design, implementation and management of software services [17].
- *Middleware* is software that is located between applications and the network layer, and is independent from operating systems. It provides the illusion of a global system in which separate components behave like a centralized system [18].
- A *method* is a description of how to conduct a process [12]. A *process* is a set of 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.
- *Techniques* consist of languages or associated modeling notations.
- *Customer value analysis* is an approach that seeks to quantify qualities that affect a customer's decision to buy a particular product. Herein, the term *value* denotes product's perceived overall benefit relative to its cost [5].

5. VIEWPOINTS

To select the appropriate viewpoints for the description of the Wise software architecture, we first have to identify the stakeholders that use and build the architecture, and their main concerns that must be covered by the selection of viewpoints. The concerns of stakeholders derive from the intention for which they need to use the architecture description. The viewpoints are described in a standard form, by means of the template presented in Table 1 (taken from [21]).

Table 1. The template to describe architecture viewpoints.

Framework element	Description
Name	The name of the viewpoint
Description	The main responsibility of the viewpoint as part of an architectural specification
Concerns	The concerns to be addressed by the viewpoint
Stakeholders	The stakeholders that are especially addressed by the viewpoint
Intention	How stakeholders use the view
Artifacts	The artifacts of the viewpoint
Constraints	What kind of information is needed in order to be able to create a view?
Functional validation	Methods how to validate the functionality of a view
Quality validation	The attributes and methods for quality validation
Language	The language and notations to be used in constructing a view


The following table summarizes the software engineering stakeholders of Wise service architecture:

Table 2. Software engineering stakeholders.

Category	Stakeholder	Description
Services	System architect	Develops a system architecture, Hw/Sw partitioning
	Service user	Uses services defined by the service architecture
	Service provider	Provides services for service users
	Service developer	Develops services for service providers
Components	Component designer	Designs components that provide services
	Component integrator	Integrates available components into services
	Component developer	Designs, implements and tests software components
Products	Product architect	Creates a product architecture
	Product developer	Develops product specific part of software, integrates components
	Product marketing	Presenting product (variable) features to customers
Software	Manager, assets manager, reuse manager	Management, costs and benefits, business, technology and reuse strategies
	Software architect	Develops software product (line) architecture
	Testing engineer	Tests software packages, integration testing
	Maintainer	Upgrades products/systems

Some of the main intentions and stakeholder concerns that must be covered by viewpoints are:

- **Getting an overview of available services and their use**
 - Describe responsibilities and context of components
- **Allocating and understanding the division of work**
 - Map services to components and vice versa
 - Map specific services to generic services

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 11 of 37
		Version: 1.06 Date: 23 Oct 03
		Status : Proposal Confid: Restricted

- Cluster the components to be developed into technology domains
- **Considering the appropriateness of service architecture**
 - What quality issues are considered
 - How qualities are attempted to be achieved with architectural styles and patterns and why these qualities are important
- **Understanding and integrating third party components**
 - Map the responsibilities of third party components into the service architecture

Even if their basic concern is similar, different stakeholders need information at different levels of abstraction and aggregation. In order to differentiate the viewpoints along those needs, the service architecture description is divided into conceptual software architecture and concrete software architecture. In both of these abstraction levels hierarchy in descriptions is used to provide the right level of aggregation for a stakeholder.

Four viewpoints are initially used at both abstraction levels: structural, behavior, deployment and development. The structural viewpoint covers the concerns related to composition of information and architectural components, whereas the behavior viewpoint considers the dynamic aspects of the architecture. The deployment viewpoint shows allocation of architectural components into physical nodes of computing and network environments. The development viewpoint shows work organization and choice of technologies mapped to services and components. Design rationale should be recorded in each view for analysis and reuse purposes [20].

Using these viewpoints the conceptual software architecture provides organization of functionality and quality responsibilities into technology domains and services in them, collaboration between the services and allocation of services into network nodes. The concrete software architecture provides hierarchical containment of concrete software components and definition of interfaces and communication protocols used between those components. The behavior of each component is described in detail and finally components are allocated to hardware resources, i.e. processors. The viewpoints of conceptual software architecture are defined in Table 3 [21]. The viewpoints of concrete software architecture are defined in Table 4 [21].


The modeling notations are based on OMG UML [23] whenever possible. Extensions or specific notations are defined when an architectural viewpoint needed in Wise cannot be expressed with UML standard set of notations. Some restrictions on types of components and relations used in the diagrams are set for the views on both the conceptual and concrete level. The use of notations is illustrated with example diagrams. The restrictions of CASE tools may influence the actual outlook of the diagrams, especially in the concrete level.

Table 3. Summary of the elements of conceptual service architecture.

Name	Conceptual structure	Conceptual behavior	Conceptual deployment	Conceptual development
Description	Mapping functional and quality responsibilities to conceptual structure.	Defining dynamic actions of and within a system.	Allocation of units of deployment to physical computing units.	Presentation of the components to be developed and acquired.
Concerns	What services and components are required? What are the responsibilities of services? How are quality requirements met?	What kinds of actions does the service architecture provide for applications? Which services do collaborate in each action? How are the actions related to each other?	Which kinds of nodes are there in a system? What services have to be in the same unit of deployment? How can services be allocated to nodes?	What services and components does the company develop and what services are acquired from third parties? Who is responsible for a service? Which standards and enabling technologies do the services use?
Stakeholders	System architects, service developers, product architects and developers, maintainers	System architect, component designers, service developers	Service users, service developers	Project manager, component acquisition
Intention	Clustering responsibilities to services. Management of commonalties and variabilities.	Finding out how middleware services are used.	Locating a service.	Project planning and management. Linking business strategies to technology strategies.
Artifacts	System context Functional responsibilities Functional structure Domain information models View design rationale	Collaboration models Table of interaction scenarios with services View design rationale	Table of units of deployment Deployment of entities View design rationale	Instantiated Business model Development model (units to be acquired and developed)
Constraints	Functional and quality requirements, architectural styles and patterns are selected for defined qualities	Co-operation with service users and system architects Incremental refinement of collaboration scenarios regarding defined qualities.	System architecture Architectural styles and patterns are selected for defined quality attributes.	Business and technology strategies, road maps. Balancing quality attributes, development time and cost.
Functional validation	Operational scenarios of service packages.	Collaboration scenarios of leaf services.	Communication scenarios of nodes/devices.	Risk analysis
Quality validation	Analysis of variation points and violations of architectural styles and patterns.	Modifiability and maintainability	Performance, security, availability, maintainability, variability in allocation	Maintainability and variability Risk analysis
Language	A set of extended UML notations.	A set of UML notations.	A set of UML notations.	Not restricted.

Table 4. Summary of the elements of concrete service architecture

Name	Concrete structure	Concrete behavior	Concrete deployment	Concrete development
Description	Decomposition at the lower aggregation level.	Behavior of individual components and interactions between component instances.	Concrete hardware and software components and their relationships.	Realizations of software components and their relationships to each other.
Concerns	What are the concrete components needed for a corresponding conceptual component? What are the interfaces needed? How do services communicate with external actors?	How does a concrete component behave and response to an event? What is the behavior of a set of concrete components?	What nodes and devices are there in a system and what they have to do? What concrete components are allocated to each node and device?	What is the realization of a service or a component? How does a service or a set of services relate each other? How could a service be configured?
Stakeholders	Component designers, service developer, product developers	Component designers, testing engineers, integrators	Integrators, maintainers	Product developers, assets managers
Intention	Verification and validation of services.	Verification and validation of co-operation of services.	Assembling and adapting services.	Maintaining asset repository.
Artifacts	Information structure Inter-component diagrams Intra-component diagrams Table of responsibilities View design rationale	State diagrams Scenarios as message sequence diagrams View design rationale	Extended deployment diagrams View design rationale	Table of component realizations Table of interface definitions Configuration models Links to asset repository View design rationale
Constraints	Conceptual structural view. Accomplishment of architectural styles and patterns with suitable design patterns.	Conceptual behavior view. Accomplishment of behavior with selected design patterns.	Conceptual deployment and development views. Accomplishment of software in physical elements.	Conceptual development view. Controlling and maintaining software qualities.
Functional validation	Simulation with generated code	Simulation with generated code, input events and tracing points.	Simulation with different allocations	-
Quality validation	Adaptability, portability and reusability	Modifiability, extensibility and maintainability	Interoperability, capacity, bandwidth	Integrability, maintainability
Language	Not restricted.	Not restricted.	Not restricted.	Not restricted.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 14 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

6. CONCEPTUAL VIEWPOINTS

6.1 OVERVIEW

At the conceptual level of architecture design, it is important to have several degrees of freedom. Too strict notations for architectural models at this stage could tie the architect's vision. The model should be more a sketching and communication tool than a means of detailed specification. Even large modifications to the basic concepts of the software architecture should be easy to make.

Conceptual Structural Viewpoint

By identifying structural elements and logical relationships among them, the Structural Viewpoint is provided mainly by UML Structure Diagrams. In particular, the following information is crucial:

- System context as networked structure diagram
- Domain models of information shared between conceptual entities, based on Class Diagrams.
- Structure and relations of conceptual entities, based on simplified Class Diagram.

Conceptual Behavioral Viewpoint

By identifying the dynamics of a system and the interactions among services, the Behavioral Viewpoint is based on Collaboration Diagrams. Special attention should be devoted to the description of collaboration between functional entities in the main use cases of the system.

Conceptual Deployment Viewpoint

Deployment Viewpoint is based on the Deployment Diagram. Deployment Viewpoint identifies the anticipated distribution needs in the system execution environment

Conceptual Development Viewpoint

The conceptual development viewpoint describes a topology model and a business model.

Business model illustrates the business relationships in the system, as well as the roles and the relationships specifically covered by the system.

The **topology diagram** illustrates *work allocation* of services and *service acquisition* i.e. the developed components and external services.

6.2 CONCEPTUAL STRUCTURAL VIEWPOINT

6.2.1 System context

The system context is perceived as the description of the networked structure (Figure 3). Services will be allocated on top of this structure. This diagram is rather informal, and aims at showing an "idea" of the environment where the system will be executed, and Hw/Sw constraints already known at the conceptual level.

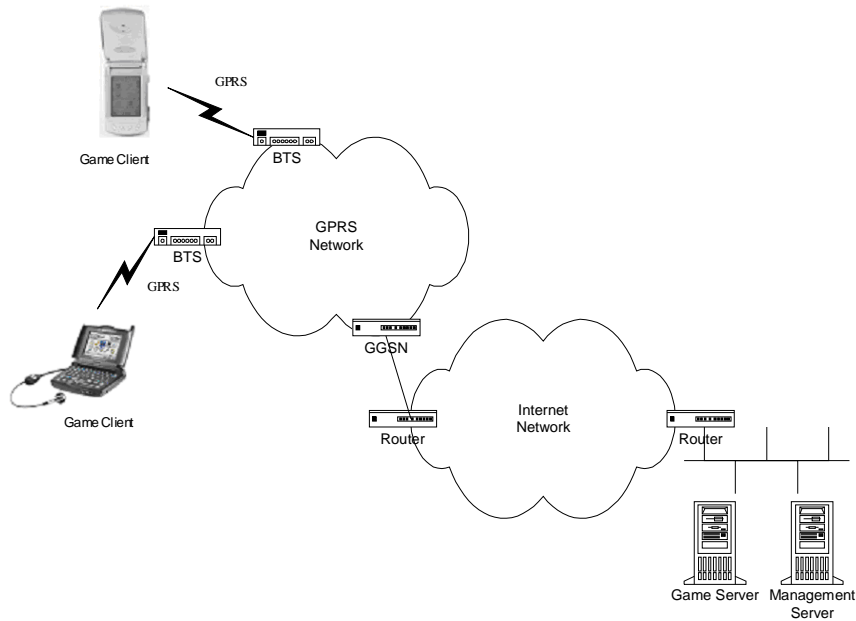


Figure 3. Example of networked structure diagram

The networked structure diagram shows the execution environment of the system under development, in terms of network resources, nodes and units present on nodes. Units can be acquired from external resources (e.g. software technologies or products), or can represent knowledge about components to be developed.

6.2.2 Domain Information models

The information models are described by using UML class diagrams. Only basic object oriented concepts should be used and implementation related concepts left out. Inheritance and aggregation can be used but methods and attributes are usually left out or kept at minimum detail (Figure 4).

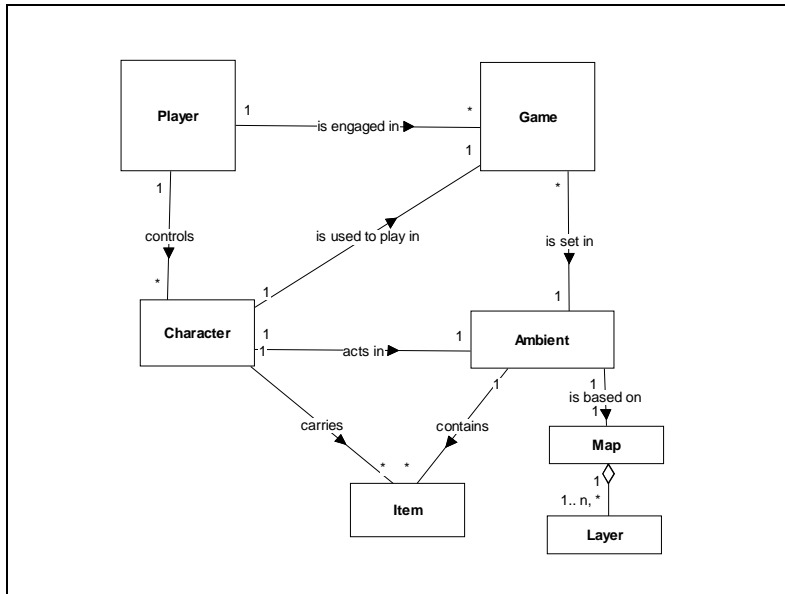


Figure 4 Example of a domain information model

6.2.3 Functional structure

Functional structure is simplified class diagram. All the architectural entities are presented with a simple classifier symbol (a rectangle). A stereotype is used to make a difference between the types of the entities (Figure 5).

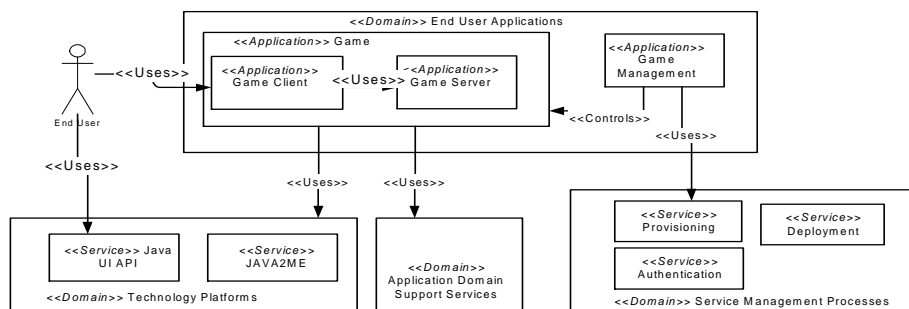



Figure 5: Functional structure of an example service.

Architect does not necessarily need to consider this type of an entity in the first drafts because the stereotype can be added later. The composition is represented with containment and other architectural relations with a stereotype of the relation. Aggregation and inheritance should not be used in order to keep the structure clear.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 17 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

The set of proposed stereotypes for conceptual entities are:

- Domain (a set of related services, i.e. a name domain)
- Application (a containment of services visible to end-user)
- Service (an end-user or middleware service)

The set of proposed stereotypes for architectural relations are:

- Uses
- Controls
- Data

The structure of a conceptual entity can be presented in a separate diagram whenever needed for clarity. The structure should be presented with minimum number of diagrams so that the overall architecture is easily visible.

The conceptual entities should be described in more detail using a separate textual description. It is preferred to use lists or table format for textual information instead of free form text, to keep the structure of the architectural description clear (Table 5).

Table 5 : Example of table format in architectural descriptions: Conceptual Element Responsibilities

Conceptual Element	Responsibilities
Game Client	Provides graphical user interface and handles the user visible subset of the game.
Game Server	Handles the game status and synchronizes the game state between different users.
Game Management	Provides common management of a game i.e. deploys and configures the needed software components and provides the usage information for the billing service.

6.3 CONCEPTUAL BEHAVIORAL VIEWPOINT

6.3.1 Collaboration diagram

The behavioral Viewpoint is based on Collaboration Diagrams. Its purpose is to identify the dynamics of a system and the interactions among services. A special attention should be devoted to the description of collaboration between functional entities involved in the main (groups of) use cases of the system. The recommended number of main use cases is around five.

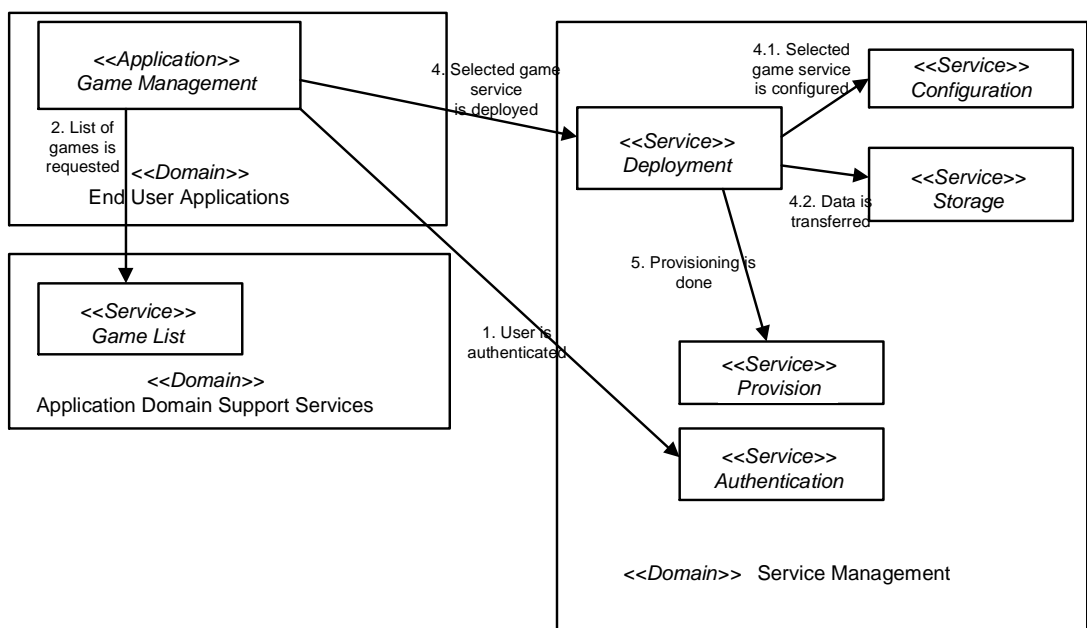


Figure 6: Conceptual collaboration in one use case of the example service.

6.4 CONCEPTUAL DEPLOYMENT VIEWPOINT

6.4.1 Deployment diagram

The conceptual deployment diagram shows the conceptual structural entities (e.g. application and services) on top of the execution nodes defined for the system context. Deployment diagram uses UML deployment diagram notation. The same service can be deployed to several different nodes (as is the Game Management in Figure 7). The detailed internal deployment of such service can be shown in a separate deployment diagram. This helps to analyze and describe the deployment needs for each service separately. It also keeps deployment diagrams within a manageable size. The relations between entities need to be shown only when they arise from the division of single conceptual entity into several (e.g. client and server) parts for the deployment or are considered especially important.

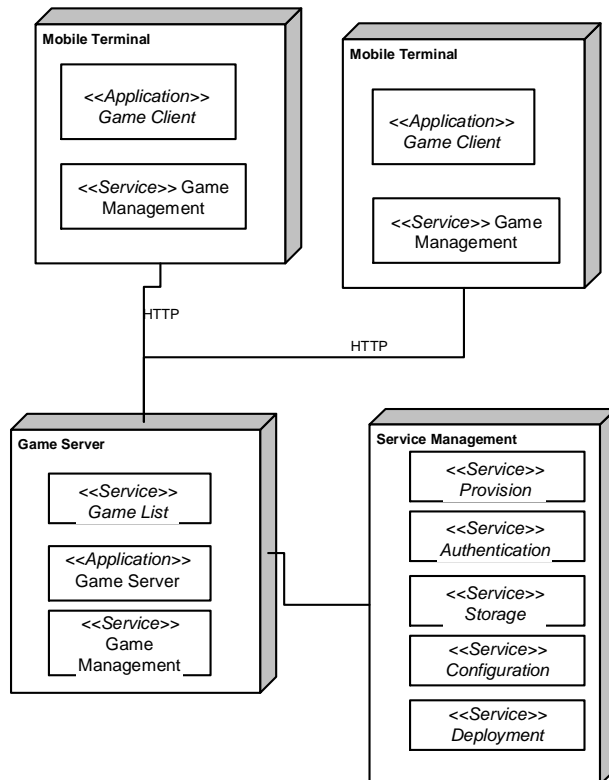


Figure 7: Example of notation for conceptual deployment

6.5 CONCEPTUAL DEVELOPMENT VIEWPOINT

6.5.1 Business Model

Modeling elements for the business model are summarized in the following. Elements shown in the figures above and adhering to OMG UML standard notation are omitted.

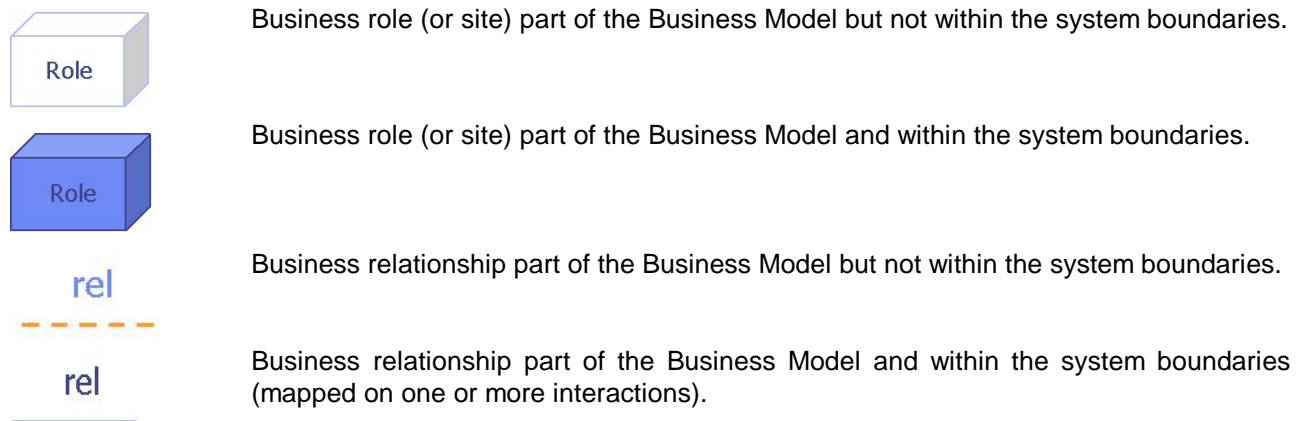


Figure 8 shows an example how to use business model notation. In the figure, the example identifies two roles played by parties (Retailer and Consumer), one business relationship implemented by the system (Ret), and one business relationship (3Pty) used by the system to interact with external roles (or external component), depicted by dashed dark lines between roles.

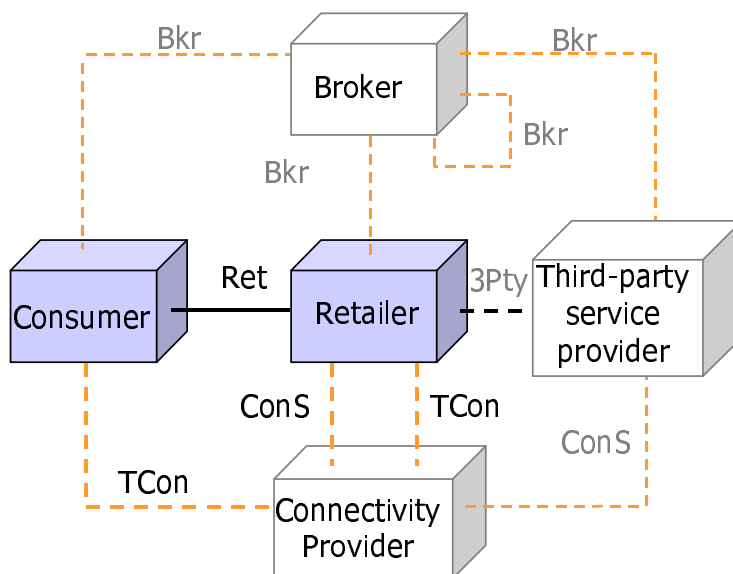


Figure 8. Example of a Business Model.

6.5.2 Topology diagram

Topology diagram is not actually a complete notation but a uniform way to describe development information on top of diagrams used in other viewpoints. Usually the viewpoint used as the basis is the functional structure.

Work allocation is described with a notation element shown in Figure 9 (a shape of a man). Work allocation element will be attached to a service or to a domain and it represents a person or a company responsible for component acquisition/development. The name of a person or a company is shown below the notation element.

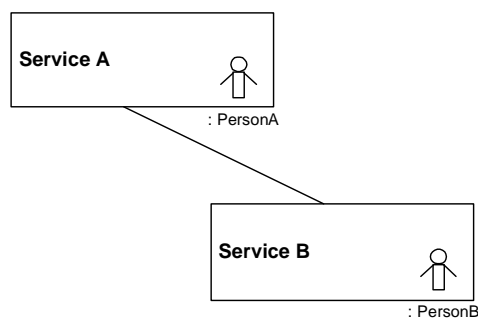


Figure 9. Work allocation element attached to a service in a topology diagram.

Component acquisition is described with a color key shown in the topology diagram example in Figure 10. The color key is inspired by traffic lights. New component (red) means a component or a service that is developed from the scratch, whereas a modified component (yellow) denotes an already developed component that is reused, but modified somehow. Commercial component (green) denotes a component developed by a third party component supplier (e.g. COTS, MOTS or OCM). Other color keys can be defined to describe more detailed acquisition information.

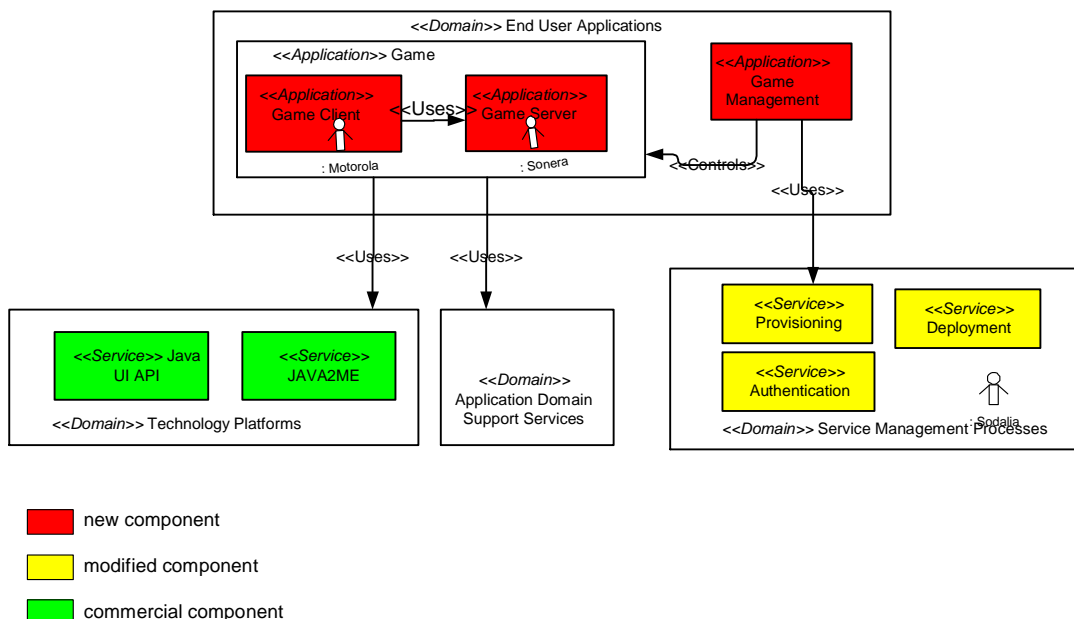



Figure 10: Example of topology diagram

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 22 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

7. CONCRETE VIEWPOINTS

7.1 OVERVIEW

Concrete Structural Viewpoint

The Concrete Structural Viewpoint is provided mainly by Class Diagrams and Component Diagrams. Their purpose is to identify concrete structural elements and relationships among them. In particular, the following information is crucial:

- Complex information managed by the system, based on the Class Diagram.
- Basic structure of computational entities, based on the Class Diagram.
- Complex structure of computational entities, based on the Component Diagram.

Of particular relevance is the representation of complex data structures in several Class Diagrams. Indeed, a computational-oriented Class Diagram identifies which classes manage which data structures, whereas details about the latter are focused in a dedicated information-oriented Class Diagram. Diagrams providing the Structural Viewpoint belong to the class (or type-level) space.

Concrete Behavioral Viewpoint

The Behavioral Viewpoint is based on Sequence Diagrams that represent the dynamics of a system and the interactions among classes and/or among components. Special attention should be devoted to the specification of cross-components and intra-components interactions. *Cross-components interactions* occur among different components and realize overall system functionality. *Intra-components interactions* occur internally to a selected component, and realize encapsulated implementation of a service offered to the external world.

The diagrams that describe the Behavioral Viewpoint belong to both the class and the instance spaces: class-level behavior is modeled by using Sequence Diagrams, and whenever required, instance-level Sequence Diagrams can show relevant example execution scenarios.

Concrete Deployment Viewpoint

The Deployment Viewpoint is based on the Deployment Diagram, which identify the actual execution environment in which a system will be operated. The execution environment can be depicted as:

1. The description of the networked structure on the top of which the system has to be installed. Details about services and technologies acquired from third parties for system execution are delegated to the Development Viewpoint.
2. The business structure (taken from conceptual development viewpoint) to which the system has to adhere, as well as the adopted business strategies.


Concrete Development Viewpoint

The Concrete Development Viewpoint shows the interfaces between the concrete components. In addition, it describes the development-time software structure and links to the assets repository with UML structural diagrams (packages and associations). Finally, the development viewpoint describes the technology layers with an informal notation.

7.2 CONCRETE STRUCTURAL VIEWPOINT

Static Structure diagrams show the static structure of a system: modules and their relationships. Among diagrams of this type we consider Class diagrams and Component diagrams, even if other diagrams (less important in the context of this work) belong to the same type, like for instance object diagrams.

We concentrate on Class diagrams and Component diagrams, as they present the same architectural viewpoint (i.e. the structural one) but on different granularities: Class diagrams define low granularity elements (the classes); Component diagrams operate at a higher level of granularity, focusing on

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 23 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

aggregations of classes and distributed interfaces (the components). As it will become clear throughout the next two Sections, these two diagrams are central to understand a distributed system, to successfully drive developers through software engineering activities, and to provide the users with detailed system documentation.

7.2.1 Essential information-oriented aspects

The information aspects can be modeled using to main elements:

- Information classes: classes modeling atomic and structured data
- Logical associations: relationships among information structures

Information is modeled as standard OMG UML Class Diagrams. They are kept separated from computational-oriented aspects.

7.2.2 Essential computational-oriented aspects

Essential elements of the computational model are:

- System components: atomic modules aggregating a collection of computational classes. Component representation should identify both the internal and the external structure of a component, in a graphical compact notation.
- Exported interfaces: interfaces offered to the external world, to make component services available to potential clients.
- Internal interfaces: operations and interfaces part of the internal structure of the component. Special attention should be dedicated in the identification (or differentiation) of language specific operations.
- Component relationships: associations existing between different components, and inside a selected component, between its classes.

Summary of notation

Modeling elements for the computational model of the Structural Viewpoint are summarized in the following. Elements shown in the preceding figures or adhering to the standard OMG UML notation are omitted.

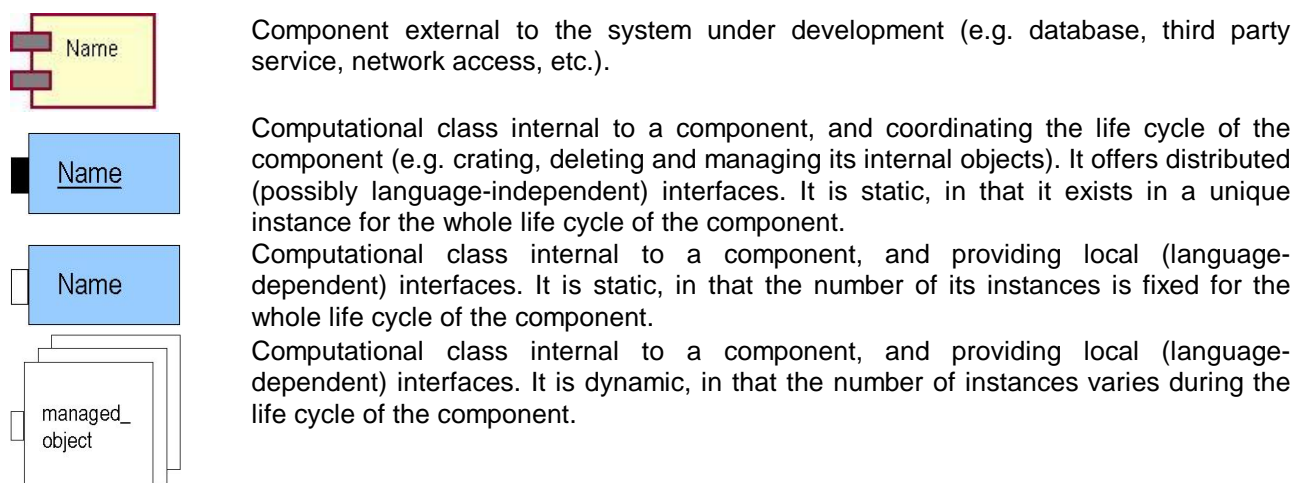


Figure 11, part (a) shows the external structure of a system component, i.e. exported interfaces, component external associations and entities external to the system. This diagram is mainly based on the UML Component Diagram, extended with external components depicted with black box interfaces.

Part (b) shows the internal structure of a system component at a lower level of detail, i.e. aggregated computational classes, internal interfaces, internal component associations, and how external functionality/associations is realized internally.

Whenever aggregated computational classes have a complex structure, they can be further detailed in additional UML Class Diagrams.

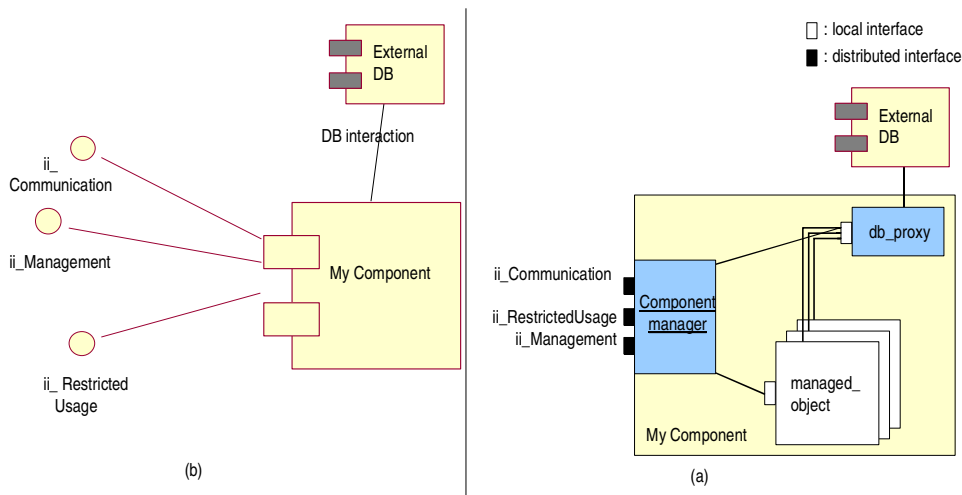



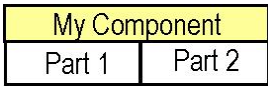
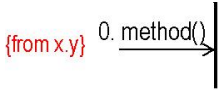
Figure 11. (a) Inter-Component Diagram, (b) Intra-Component Diagram.

7.3 CONCRETE BEHAVIORAL VIEWPOINT

Interaction diagrams in UML (i.e. Sequence and Collaboration diagrams) show the runtime behavior of a system. This means to model how components exchange messages and invoke one another interfaces and operations to achieve the overall system functionality.

Summary of notation

Modeling elements for the Behavioral Viewpoint are summarized in the following. Elements adhering to OMG UML standard notation are omitted.

- 
Black box component (for which internal details are omitted).
- 
Component detailed in its constituent parts (interfaces and internal objects) to explain how an interaction scenario is served internally by the component itself.
- 
Invocations arriving from (or directed to) “nowhere” provide horizontal modularization of scenarios. They can be decorated with the name of the scenario (X) and its subpart (Y) for documentation.

When a system is complex, behavior can be organized on two abstraction levels, namely inter-component behavior (depicting overall interactions of a system as a whole) and intra-component behavior (focusing on single components and showing how overall functionality is served internally by component members).

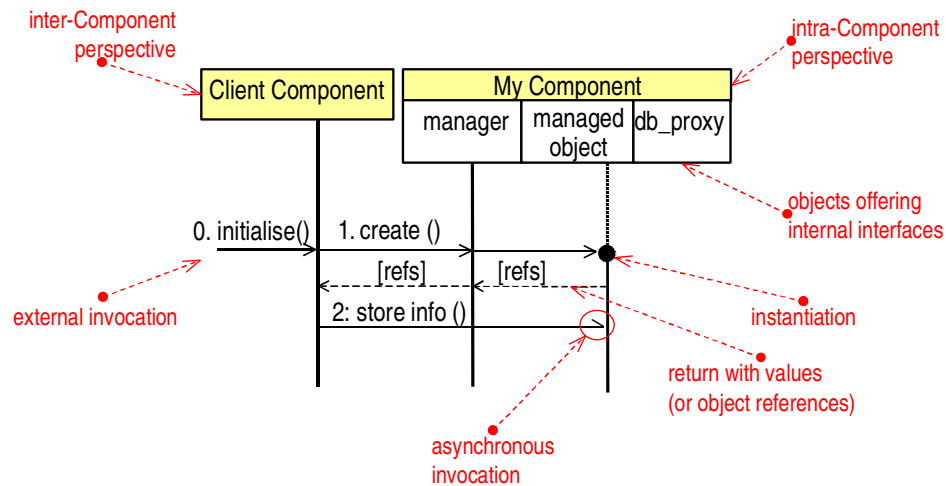


Figure 12. Notation for Intra- and Inter-Component interactions

To add sequencing information, we basically rely on sequence diagrams, which can be modeled at both class and instance levels.

Accordingly, *Inter-Component Sequence Diagrams* depict scenarios of interactions between different components, to achieve overall functionality. To this aim, components (and possible external elements) are depicted as black box elements, without entering the details of how interactions are supported internally by each component.

Similarly, *Intra-Component Sequence Diagrams* depict how scenarios are realized inside a component. Therefore, in this diagram components are detailed in their parts, i.e. constituent objects (if necessary) and exported interfaces (if multiple).

Figure 12 summarizes the main aspects in modeling interactions in Intra-Component Sequence Diagrams. We can observe that the component, on which the diagram is focused, is detailed in its interfaces, whereas the external (Client) component being the source of interaction is depicted as a black box element.

Guidelines for the use of Sequence Diagrams are:

- Diagrams are needed only for those interactions, which are particularly crucial, critical or complicated.
- Diagrams should represent composite objects and multiple interfaces.
- Diagrams can provide vertical modularization, i.e. intra-component and inter-component perspectives.
- Diagrams can provide horizontal modularization, i.e. represent fragmented scenarios if fragments show interactions recurring in multiple scenarios, or if a scenario is particularly complicated.
- Diagrams should represent adherence to standards or recurring interaction patterns, possibly modeled at class level.
- Diagrams can focus on input to and output from the system, to drive interface development. The representation of return parameters on return interaction arrows is crucial.

7.4 CONCRETE DEPLOYMENT VIEWPOINT

7.4.1 Deployment diagram

This diagram shows which interactions implement the identified business relationships, which connections are needed, possible security or contractual requirements, etc.

For those systems implementing service architectures, interface-level standards can be used to define business relationships in terms of technical interactions, so that by mapping a standard on the chosen business model, system compliance is automatically achieved.

Figure 13 shows an example of a networked structure of on top of which system components are to be deployed. This diagram also details the number and types of nodes for each business role, and how/which cross-component interactions realize the business relationships modeled in the conceptual development view.

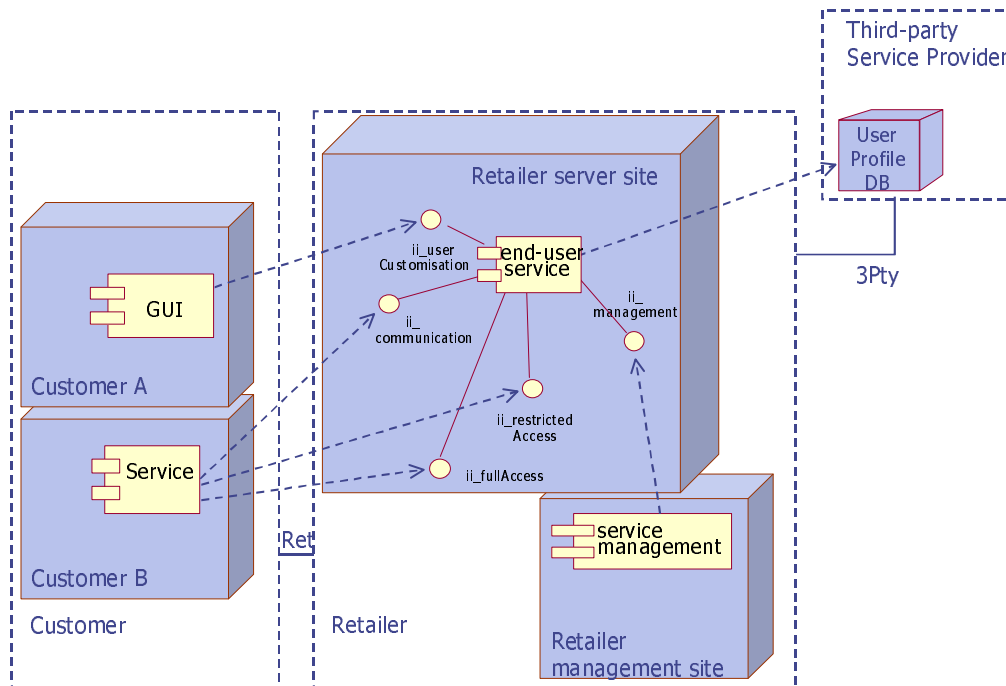


Figure 13. Example of Deployment of Components.

7.5 CONCRETE DEVELOPMENT VIEWPOINT

7.5.1 Interfaces

The interfaces between developed concrete components are presented in the format of a table. Table 6 is an example of such a table. Interface hierarchy can have several levels i.e. interfaces can be bundled into larger interface entities. Interfaces between component developed by single stakeholder can also be shown along with concrete structural viewpoint. More detailed explanation of the interface description model is in the chapter 7.

Table 6 : Example of abstract interface definition

Interface	Responsibility	Operation
srvMgm_ITF: asynchronous EJB (RMI/IIOP)	Allows the access to Service Management Services, such as authentication and authorisation of a user, user profile management and accounting.	authenticate authorize userProfile accountRequest subscribe unsubscribe
srvMgm2_ITF: asynchronous EJB	Allows the message handling from the SelfSubscription Service	<i>requestNotification</i>


The Implementation level “Interface definitions for a specific technology” are described in a separate diagram.

7.5.2 Development structure

The development-time software structure and links to *assets repository* (if present) can be shown with UML package diagrams.

7.5.3 Technology layers

The technology layers can be described with informal notation.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 28 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

8. INTERFACE DESCRIPTION MODEL

In order for a service to interact with other services, there must be technical conventions for standardising interactions. This standardisation includes messaging formats, interaction definitions, properties of the interactions (security, performance) etc. Concept of interfaces is often used to model interactions. Interoperability and composability requirements of the wireless services lead to the fact that the standardised way to describe service interfaces is required. Currently, constraints in description models and in their implementations are restricting interoperability between service providers. In the following, a suggestion for the standardised interface description is proposed.

Interface description model has two levels. The first level illustrates the interfaces from the architectural point of view, describing the responsibilities of the interface. The second level is a detailed description of the transformation of the interfaces, i.e. how the interfaces are mapped to the implementation. The idea behind the levels has been in the description of the whole service. The purpose of the service description is to describe the whole service using XML, so that the tools can automatically understand the description. WSDL (Web Services Description Language) was firstly examined to be used in the service description. However, WSDL was not adequate for this purpose, because it describes services only as endpoints and messages.


The architectural level interface description is an important part of the service description. The interface description should obey the same principles as the service description, so that it can be included as a part of service description when needed. To enforce this, eXtensible Markup Language (XML) implementation of the proposed interface description is also presented. The transformation level description reveals the alternative implementations of the interfaces.

The traditional table format is not adequate for the interface description for several reasons. Firstly, the table format is not flexible. There may be different kind of information available for different interfaces, when the description of these is difficult by using the same table. In addition, the description of different hierarchy level is difficult by using a table format. Secondly, the table format does not allow the effective information search when retrieving information from the service or the interfaces. Thirdly, the table format does not enable the different information presentation, i.e. different views on the information. Therefore, more powerful description technology for interface description is required.

The chosen technology, eXtensible Markup Language (XML), is the World Wide Web Consortium's (W3C) recommendation for a meta-markup language [26]. XML provides a mechanism for describing the document content, structure and meaning. It also enables platform-independent data exchange between applications [27]. XML was chosen because of its extensibility and application independency. XML is a non-proprietary format and is not encumbered by any sort of intellectual property restriction. Any tool that understands XML format can be used to handle XML documents.

8.1 THE ARCHITECTURAL LEVEL INTERFACE DESCRIPTION

The architectural level interface description is a specification for the architectural level of the service. Graphically, the interfaces can be described using the external component diagram

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 29 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

that illustrates both required and provided interfaces of the service. However, the graphical presentation is not informative enough, so more detailed description is required.

The interface description of the architectural level consists of the following elements: interface name, bundle, communication type, list of implementations, responsibility and operation. Interface name should be well defined and describe the use or the purpose of the interface. An interface should be able to be composed from different interfaces. Therefore, an interface may be a part of an interface bundle that is a collection of interfaces. The interface composition can have several hierarchical levels, but these are restricted here because of the description format. The name of the possible bundle should be defined within the interface name. Interface communication describes the type of communication that occurs through the interface. For example, in synchronous communication a receiving object must be ready to communicate with the sending object at all times, whereas in asynchronous communication a receiving object can retrieve messages at its convenience. Implementation in this context means the technology that the interface supports. There may be several implementations for the interface, so the used technologies are listed here. Responsibility describes the interface's responsibility, i.e. what are the assignments that the interface is responsible of. Operation is a list of operations that interface enables. The operations are introduced here only by name. The more detailed description is given in an interface transformation description.

Table 7 displays the introduced elements of the interface description. The interface field in Table 7 can also include a reference element that is an optional element that will be used when necessary. The reference element is used to make a reference to architectural design document, where the information about the interface or the interface bundle is available. The reference should also reveal the architectural view that is used to define the target of the reference.

Table 7. Architectural level interface description.

Interface	Responsibility	Operation
Name of the interface (the name of the possible interface bundle): communication type List of implementation technologies (i.e. variants). Reference (reference to the architecture design document)	Description of the interface's responsibility	Operation name

8.2 THE TRANSFORMATION LEVEL INTERFACE DESCRIPTION

The transformation level interface description describes how the interfaces are transformed from the architectural design Level to the implementation Level. Each interface is described separately. In addition, there may be several variants for each interface. In this context, a variant is an alternative implementation of the interface. Each variant should be described using the following table (Table 8).

The interface description at the transformation level consists of the following elements: implementation, method, responsibility, parameters, return and exceptions. The name of the interface is the caption of the table; also the implementation technology is mentioned in the caption to identify the variant. Implementation in this context means the technology that the interface supports. Method element must always correspond to the operation from the architectural level description. In this context, a method is an implementation of an operation. The primitive for the method is one of the message transmission primitives defined in Web Services Description Language (WSDL) [25]: one-way, request-response, solicit-response or notification. WSDL describes network services as a set of endpoints operating on messages. One-way transmission means that the endpoint receives a message. In request-response transmission the endpoint receives a message, and sends a correlated message, whereas in solicit-response transmission the endpoint sends a message, and receives a correlated message. In notification transmission the endpoint sends a message. Responsibility element describes the purpose and responsibilities of the method. Parameter responds the "part name" in WSDL. Messages consist of one or more logical parts. Each part is associated with a type from some type system, such as XSD, using a message-typing attribute. Thus, a parameter type is a data type definition that is relevant for the exchanged message. The return element imposes the return-value of the method. Exceptions element describes the possible exceptions in communication. Table 8 displays the introduced elements in the table format.

Table 8. Transformation level interface description.


Interface name: technology				
Method	Responsibility	Parameter	Return	Exceptions
Method name: transmission primitive	Description of method's responsibility	Parameter name: type	The return-value of the method	Description of exceptions in communication

8.3 THE IMPLEMENTATION OF THE INTERFACE DESCRIPTION

In the following, the interface description is transformed to the XML format. An XML document consists of semantic tags (elements) that break a document into parts and identify the different parts of the document. The extensibility and self-describing nature of XML means that users can define their own set of markup tags. These tags must be organized according to certain general principles of a Document Type Description (DTD), which specifies the rules for the structure of a document. XML does not include any formatting instructions, but the formatting can be added into documents with style sheets [24].


XML allows an easy data retrieval from the whole service description. Separate XSL stylesheets allow the creation of different views from the XML data. XSL (Extensible Style Language) is a style language for presenting structured content - i.e. styling, laying out and paginating the source content onto some presentation medium, such as a web browser [24]. With the help of a stylesheet, the interface description can be easily viewed in desired format. In the same way, the unnecessary information can be filtered away when needed.

Figure 14 shows the developed XML based document template of the interface description. The document template is the base form for the interface description, including all the required elements and attributes. The elements and attributes are highlighted with red color. The instructions for the use of the XML template are placed in the template between square brackets.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 31 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

The XML based interface description consists of interface model and the description of the both provided and required interfaces. The interface model is usually a picture of external component (or service) interfaces. The model element includes the name attribute, and has a child element called image. The image element also has the name attribute for the name of the image and src attribute for the source of the image. The image element may also have a child element called caption for inserting the caption for the image.

Interfaces, both provided and required, are described with interface element that has the name and bundle attributes and child elements, such as responsibility, communication, reference, operations and variant. Reference element has a child element called target that has a href attribute for the location of the referenced document. The operations element can have child elements called operation one to many. Interface element may have one to many variant elements, depending on the amount of the different interface implementations. The variant element has an attribute called technology and a child element called methods. The methods element can have child elements called method one to many. The method element has a name attribute and the child elements such as type, responsibility, parameters, return and exceptions. The parameters element can have one to many parameter-child elements. . The parameter element has the name and type attributes.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 32 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

```

- <interface_description>
- <model name="[external component interfaces]">
  - <image name="[The name of the image.]" src="[The source of the image.]">
    <caption>[The caption of the image.]</caption>
  </image>
</model>
<description>[Free-form textual description of the external interfaces of the service.]</description>
- <interfaces>
- <provided>
  - <interface name="[The name of the interface.]" bundle="[The name of the possible interface bundle that this interface is part of]">
    <responsibility>[Description of the responsibilities of the interface.]</responsibility>
    <communication type="[Communication type description]" />
  - <reference>
    <target href="[Location of the target document]">[The target document]</target>
  </reference>
  - <operations>
    <operation name="[The name of the interface operation]" />
  </operations>
  - <variant technology="[The implementation technology of the variant.]">
  - <methods>
    - <method name="[The name of the method]">
      <type>[Transmission primitive]</type>
      <responsibility>[Description of the responsibilities of the method.]</responsibility>
    - <parameters>
      <parameter name="[Name of the parameter]" type="[Parameter type]" />
    </parameters>
    <return>[Return value of the method]</return>
    <exceptions>[Exceptions in communication]</exceptions>
    </method>
  </methods>
  </variant>
</interface>
</provided>
- <required>
  <!-- the required interfaces are described the same way as the provided interfaces -->
</required>
</interfaces>
</interface_description>

```

Figure 14. XML template of the interface description.

8.4 AN EXAMPLE OF THE USE OF THE MODEL

In the following, the interface description model is demonstrated using a sample service. The sample service, Service Management Component, is one of the WISA basic services (see WISA reference architecture). The service addresses the following functional area: authentication and authorization of users, user profile management, self-subscription management and accounting and mediation/rating. The Service Management Services are accessible through an interface that is a facade between service management server and application servers. Figure 15 shows the external interfaces of the Service Management Component. The SrvMgm_ITF interface groups together the methods required to use the services. Service Management Component requires a SSS_ITF interface to receive notifications concerning the subscription

made by a user for a service. Administration interface is used to remotely administer the component. This interface is unexposed to the user of the service and therefore it is not concerned here.

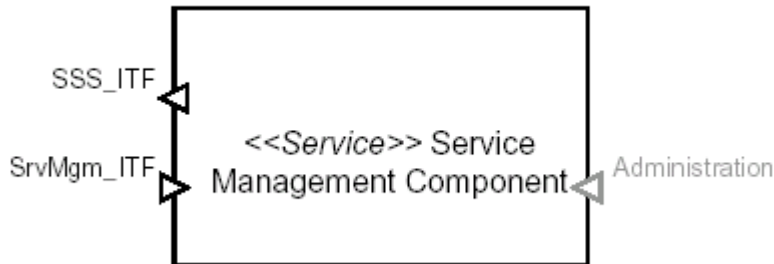


Figure 15. External interfaces of the Service Management Component.


In the following tables (Table 9 and Table 10), one of the interfaces of the sample service is described using the table format:

Table 9. An example of the architectural level interface description.

Interface	Responsibility	Operation
srvMgm_ITF: asynchronous EJB (RMI/IIOP)	Allows the access to Service Management Services, such as authentication and authorisation of a user, user profile management and accounting.	authenticate
		authorize
		userProfile
		accountRequest
		unsubscribe

Table 10. An example of the transformation level interface description.

srvMgm_ITF: EJB (RMI/IIOP)				
Method	Responsibility	Parameters	Return	Exceptions
authenticate: request-response	Wraps a call to the Authentication Service. The method returns true if the credentials provided are correct.	user: String password: String	Boolean	authentication failed
authorize: request-response	Wraps a call to the Authorization Service.	user: String service: String	Boolean	authorization failed
userProfile: request-response	Wraps a call to the User Profile Service. The returned Map is a key-value association that describes the profile of a user.	user: String	Map	getting userProfile failed
accountRequest: request-response	Wraps a call to the Accounting Service.	id: String timestamp: Date values: Object []	Accounting Reply	can't execute accountingRequest with

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 34 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

				sessionId
subscribe: request-response	Reads the request type.	subscriptionId: String user: String reqType: String service: String	void	Error at message arriving
unsubscribe: request-response	Reads the request type.	subscriptionId: String user: String reqType: String service: String	void	Error at message arriving


In the following, the XML template is applied to describe an interface of the sample service (Figure 16).

```


- <interface name="srvMgm_ITF">
  <responsibility>Allows the access to Service Management Services, such as
  authentication and authorisation of a user, user profile management and
  accounting.</responsibility>
  <communication type="asynchronous" />
  - <operations>
    <operation name="authenticate" />
    <operation name="authorize" />
    <operation name="userProfile" />
    <operation name="accountRequest" />
    <operation name="subscribe" />
    <operation name="unsubscribe" />
  </operations>
  - <variant technology="EJB (RMI/IIOP)">
    - <methods>
      - <method name="authenticate">
        <type>request-response</type>
        <responsibility>Wraps a call to the Authentication Service. The method
        returns true if the credentials provided are correct.</responsibility>
        - <parameters>
          <parameter name="user" type="String" />
          <parameter name="password" type="String" />
        </parameters>
        <return>Boolean</return>
        <exceptions>authentication failed</exceptions>
      </method>
      + <method name="authorize">
      + <method name="userProfile">
      + <method name="accountRequest">
      + <method name="subscribe">
      + <method name="unsubscribe">
    </methods>
  </variant>
</interface>
</provided>

```

Figure 16. The XML description of an interface of the sample service.


	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 35 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

The resultant document can be viewed with a web browser using a style sheet that formats the XML data e.g. to the table format.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 36 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

9. REFERENCES

1. Bachmann, F. and Bass, L. Managing Variability in Software Architectures. Proceedings of SSR'01, ACM. 2001. Pp. 126 - 132.
2. Bass, L., Clement, P. and Kazman, R. Software Architecture in Practice. Addison-Wesley. 1998.
3. Bosch, J. Design & Use of Software Architectures. Addison-Wesley. 2000.
4. Buschmann, F., Meunier, R. and Rohnert, H. Pattern-oriented software architecture, a system of patterns. John Wiley & Sons. 1996.
5. 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.
6. Gamma, E. Design patterns: elements of reusable object-oriented software. Addison-Wesley. 1994.
7. IEEE Computer Society, IEEE Recommended Practice for Architectural Descriptions of Software-Intensive Systems. IEEE Std-1471-2000.
8. Johnson, R., Foote, B.: Designing Reusable Classes, Journal of Object -Oriented Programming, 1 (2), 22-5.
9. 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.
10. Kang, K. C., Sholom G. C., Hess J. A, Novawk W., and E. Peterson A. S., K. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, 1990.
11. Niemelä, E. A component framework of a distributed control systems family. VTT Publications 402, Espoo, Technical Research Center of Finland. 1999.
12. Kronlöf, K. Method integration: concepts and case studies. John Wiley & Sons. 1993.
13. Perry, D. and Wolf, A. Foundation for the Study of Software Architecture. SIGSOFT Software Engineering notes, vol. 17, No. 4, 1992, Pp. 40 - 52.
14. Shaw, M. and Garlan, D. Software Architecture. Perspectives on an Emerging Discipline. Prentice Hall. 1996
15. Sodhi, J., Sodhi, P., Software Reuse, Domain Analysis and Design Process. McGraw-Hill, 1999.
16. Szyperski, C. Component Software. Beyond Object-Oriented Programming. New York: Addison Wesley Longman Ltd. 1997.
17. TINA Consortium Service Architecture specification. <http://www.tinac.org>
18. Emmerich, W., Engineering distributed objects. John Wiley & Sons, Ltd., 2000.
19. Buschmann, F., Maunier, R., Rohnert, H., Sommerlad, P., Stal M., A System of Patterns – Pattern-oriented Software Architecture. Wiley 1996.

	Architectural Guidelines Deliverable ID: D4 (Part A)	Page : 37 of 37
		Version: 2.0 Date: 23 Oct 03
		Status : Proposal Confid. : Restricted

20. Matinlassi, M., Niemelä, E., Dobrica, L. Quality-driven architecture design and quality analysis. A revolutionary initiation approach to a product line architecture. 129 p. + 10 p.
21. Purhonen, A., Niemelä, E., Matinlassi, M. Views of DSP software and service architecture. Submitted to Journal of Systems and Software, 31 p.
22. Jacobson, I., Griss, M., Jonsson, P., “Software Reuse. Architecture, Process and Organization for Business Success”, Addison Wesley, 1997.
23. OMG (Object Management Group), “OMG Unified Modeling Language Specification”, Version 1.4, Sep. 2001. On-line at <http://www.omg.org/uml>.
24. Harold, E. 1999. XML Bible. Foster City, USA: IDG Books WorldWide, Inc. 1015 p. ISBN: 0-7645-3236-7.
25. World Wide Web Consortium 2001- Web Services Description Language (WSDL) 1.1. Available: <http://www.w3.org/TR/wsdl>.
26. World Wide Web Consortium 2001. Extensible Markup Language (XML). Available: <http://www.w3.org/XML/>.
27. Walsh, N. 1998. A Technical Introduction to XML. InterCHANGE, Vol. 4, Issue 2. Pp. 17-26.