# *A Template for Documenting Software and Firmware Architectures*

*Version 1.3, 15-Mar-00*

Michael A. Ogush, Derek Coleman, Dorothea Beringer

Hewlett-Packard Product Generation Solutions
mike_ogush@hp.com
derek_coleman@hp.com
dorothea_beringer@hp.com

**Abstract**

This paper defines a template for producing architectural documentation. Two different kinds of architectural documentation are identified: an *architectural overview* and an *architecture reference manual*. The template specifies a common structure for both kinds of document and illustrates its use with examples. The focus of the template is on the logical view of a system including system purpose, system context and interface, structure of the system, and dynamic behavior of the system. Other system views like process view, physical view, or conceptual framework view of the problem domain are also integrated. The template is intended for use in product development for defining the architecture of software and firmware projects.

**Key words:** software architecture, document template, components, interfaces, scenarios.

# Table of Content

## 1. Overview

In recent years a realization has grown of the importance of software architecture. According to Bass et al [1], the software architecture of a system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. The IEEE recommendation [2] defines an architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. Software architectures are important because they represent the single abstraction for understanding the structure of a system and form the basis for a shared understanding of a system and all its stakeholders (product teams, hardware and marketing engineers, senior management, and external partners).

This paper tackles the problem of how an architecture should be documented. It is a meta-document that defines a template for producing architectural documentation. As such it defines how to document purpose, concepts, context and interface of a system, how to specify system structure in terms of components, their interfaces, and their connections, and how to describe system behavior. Although the paper uses the term software architecture throughout, the template has proven to be also applicable to firmware architectures with little or no modification.

The structure and content for an architectural description is given in section three of this paper. Each subsection of section three describes the form and content of a section of an architecture document. Thus section 3.1 of this template describes what information should be given in the Introduction section of an architecture document; section 3.2 describes the Purpose section of an architecture document etc. Most explanations are accompanied by examples taken from a (fictitious) architecture document for CellKeeper network management system [3].

A summary of the structure of an architecture document is given in appendix A. Appendix A is the ideal starting point for everybody who wants to get a quick overview of the various elements of the architecture template.

Section two of this paper discusses the different contents, purposes and readerships for architectural documentation and how that affects the use of the template. Appendix B shows how the architecture template presented here relates to the IEEE Draft Recommended Practice for Architectural Description. Appendix C contains a glossary of important terms.

## 2. The Role and Content of Architectural Documentation

***Architectural overview and architectural reference manual***

The template can be used to produce two different kinds of architectural documentation, an *architectural overview* and an *architectural reference manual.*

An **architectural overview** is aimed at providing a shared understanding of the architecture across a broad range of people including the developers, marketing, management and possibly potential end-users. An architectural overview is ideally produced early in the development lifecycle and serves as the starting point for the development. An architectural overview should be at a high level of abstraction. All the major functionalities and components of the architecture should be described but the descriptions may lack detail and precision as they often use natural language rather than formal notations.

An **architectural reference manual** describes an architecture in a detailed and precise manner. Unlike an architectural overview, an architectural reference manual is not written at one particular point in time. Rather a reference manual should be a living document that is constructed collaboratively by the development team as the development proceeds. As it develops the reference manual can be used to track progress of the software development and for assessing the impact of proposed requirements changes. When complete the reference manual is the basis for system maintenance and enhancements. The reference manual should be updated as changes occur so it always reflects the actual architecture.

A reference manual needs to be complete in the sense that every facet of the architecture should be documented. Although a reference manual will tend to be detailed and technical in nature the level of abstraction and precision is likely to vary across the architecture and across different projects. The architecture within very large grained components should be documented in separate architectural reference manuals for the components. The level of completeness, formality, detail and precision used for any particular aspect of the architecture will depend on the associated technical or business risk.

### *Scoping*

Not every architecture documentation requires all the sections described in this template. Appendix A gives a summary of all the sections and shows which ones are optional. Yet even for the mandatory sections the amount of information documented not only varies between architectural overview and architectural reference manual, but also from project to project. The introduction section of the architecture document lists the stakeholders and their concerns. An architecture document is complete as soon as the concerns of the stakeholders are met.

### *Views covered by the architecture template*

The template has been structured according to the 4 views of the 4+1 view model of Kruchten [4]: the *logical view* is modeled in the structure section and the dynamic behavior section, the *process view*, the *physical view* and the *development view* are modeled in the other views section.

For each view the structure of the components and the dynamic behavior, i.e. scenarios showing the interactions of the components, are modeled. For the process, physical and development view this is done in the process, physical, or development view sections. For the logical view, it is split up into two sections: the structure section and the dynamic behavior section. Of course, the dynamic models in the different views must be consistent. They can be looked at as the integrating force between the various views.

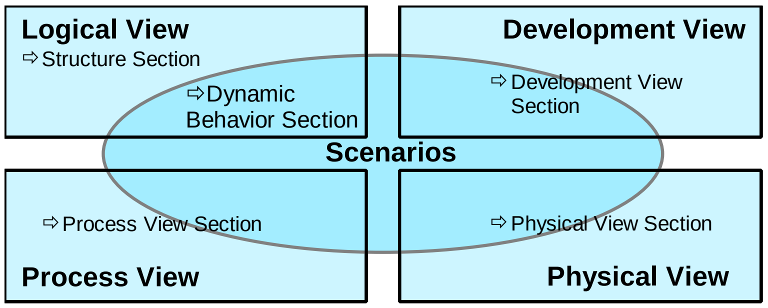


*Figure 1: 4+1 View Model*

# 3. Template for Architectural Documentation

Sections 3.1 to 3.7 describe the structure and content of an architecture document. For each section of such a document it provides a description of the structure, an explanation and, in all non-trivial cases, an example. The following notational convention is used:

- Syntactic *structure* of each part of the section is shown diagrammatically in a figure with yellow background (e.g. Figure 2), or as a table (e.g. Table 1).
- The normal text gives *explanations* of the content of each section.
- *Examples* are shown in figures with a double edge (e.g. Figure 4).

Figure 2 shows the top-level structure of an architecture document. Each box names a section of the document and briefly describes its purpose.

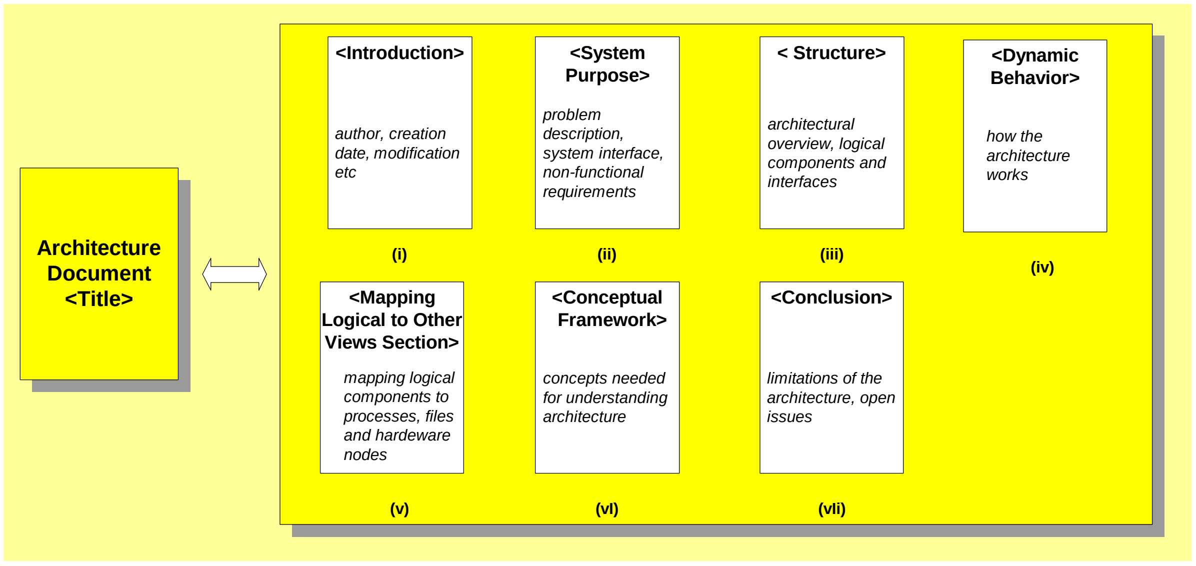

Architecture Document <Title>
<Introduction>
author, creation date, modification etc
(i)
<System Purpose>
problem description, system interface, non-functional requirements
(ii)
< Structure>
architectural overview, logical components and interfaces
(iii)
<Dynamic Behavior>
how the architecture works
(iv)
<Mapping Logical to Other Views Section>
mapping logical components to processes, files and hardware nodes
(v)
<Conceptual Framework>
concepts needed for understanding architecture
(vi)
<Conclusion>
limitations of the architecture, open issues
(vii)

*Figure 2: Sections of an architecture document*

The following chapters discuss the structure and content of each section of the architecture document.

## 3. 1 Introduction Section

The introduction section identifies the architecture and its stakeholders, and it records the creation and subsequent modification to the architecture documentation. Details to be captured include:

- *name of the architecture*
- *architecture design team and author of the architecture document with information on who to contact to provide feedback for the architecture and its documentation*
- *creation and modification history*
- *audience and purpose*
- *selected viewpoints*
- *related documents*

As part of the introduction, the architecture document should state whether the document is an architectural overview or a reference manual, who the stakeholders and the intended readers are, and what the intended purposes of the document are. It should also record the relationship to any other documents associated with the development of the software, e.g. System Requirements Specification, System Architecture Specification, Design Specification, Internal Reference Specification, etc. Optionally[1], the selected viewpoints (see appendix B) can be listed together with the stakeholders, and the issues addressed by each viewpoint, and a list of the sections of the architecture document containing descriptions and models for the selected viewpoints.

When there is a single product associated with the architecture this section may optionally contain information regarding the project/product using the architecture like project name, release date, project team, or product team.

## 3.2 System Purpose Section

This section documents the system in terms of its purpose and the problem that it solves. It describes the

- context in which the system will be used and the problem(s) that it solves,
- the services, i.e. functionality, that the system provides at its interfaces, and
- the qualitative characteristics of these services.

The system purpose section provides a black box view of the system to be designed. The system must satisfy the requirements as defined in any valid system requirements document. If for any reason there are requirements which are not met by the system, then these must be explicitly recorded. Normally, the system purpose section gives a summary of the specifications concerning context, system interface and non-functional requirements contained by any system requirements specification documents. [2]

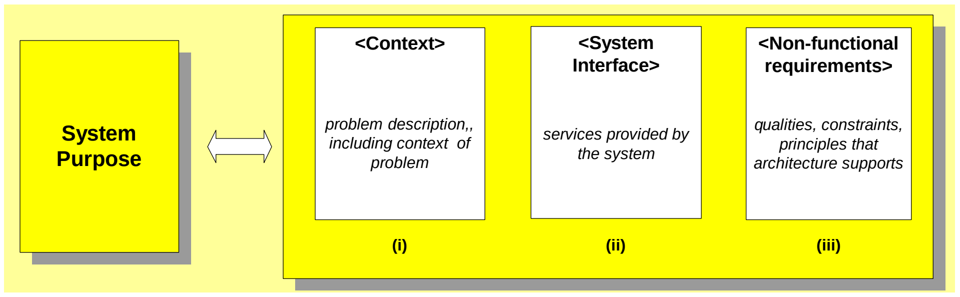

System Purpose
<Context>
problem description,, including context of problem
(i)
<System Interface>
services provided by the system
(ii)
<Non-functional requirements>
qualities, constraints, principles that architecture supports
(iii)

*Figure 3: System purpose section*

### 3.2.1 Context Section

This section briefly and informally describes the context of the system and the problem that it solves.

The aim is to provide an introduction to the system that is accessible to non-domain experts. The **problem description** should enumerate the key entities involved with the system and how the system provides value to them. The focus of this section is on the entities interested in and communicating with the system, and on the roles of these entities, not on the system itself.

[1] Needed for conformity with IEEE Recommended Practice for Architectural Description

[2] Depending on purpose and audience of the architecture documentation, the system purpose section can simply contain references to the appropriate sections in requirements specification documents,
- if a system requirements specification contains the same information or models as required for the sections "Context", "System Interface" and "Non-Functional Requirements",
- if such system requirements documentation is maintained and updated during the life of the system,
- and if these documents are readily available to the audience of the architecture documentation.

In order to describe the problem solved by the system it is necessary to delineate the boundary between the system and its environment. It is sometimes also helpful to show the larger context of which the system is part of, inclusive associations and data flows between other systems. The **context diagram** can be one of the following:

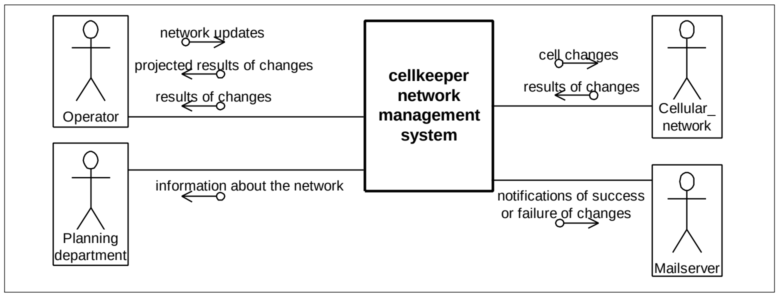
- an *object or class diagram* showing the system under consideration, other systems interacting with this system or otherwise valuable for understanding the context, and all important associations,
- a *high-level use case diagram* showing the system, its actors[3], and the most important use cases,
- a *data flow diagram* showing the data and control flows between the system and other entities in its environment (as in Figure 4).

***Example***

**CellKeeper Purpose and Context**

Mobile phones are controlled by a network of cells that manage the call traffic on radio frequencies. These cells are responsible for covering a geographic area, knowing about adjacent cells, and handing over calls to other cells when the signal quality is reduced due to movement of the mobile phone.

CellKeeper is an application that can manage the configuration of a network of cells. E.g., when there is an increased need for capacity of the network, a operator can interact with CellKeeper to split an existing cell into multiple cells. Also, CellKeeper can download new parameters to an individual cell (cell frequencies, exclusive use of cell by phones for emergency purposes only, etc.).


Operator
network updates
projected results of changes
results of changes
Planning department
information about the network
cellkeeper network management system
cell changes
results of changes
Cellular_ network
notifications of success or failure of changes
Mailserver

Context Dataflow Diagram of the CellKeeper System

*Figure 4: Example CellKeeper: system context with a data flow diagram*

### 3.2.2 System Interface Section

The system interface section documents the services that the system provides in terms of responsibilities. Often the system interface may be organized into a set of sub-interfaces, each sub-interface corresponding to a distinct usage of the system, e.g. there may be specific interfaces for system configuration, for normal system use, and for system management.

Interfaces may be defined at varying levels of detail and precision. In an architectural overview document the system interfaces will be described at a very high level of abstraction with optionally simply listing individual system operations or use cases[4] (see example in Figure 5). In an architectural reference manual

[3] An actor is any active entity in the environment of the system that interacts with the system.

[4] A use case consists of several steps and interactions between one or several external actors and the system in order to achieve a specific goal. If on a given abstraction level the steps of a use case are not further decomposed, then the steps are considered as being system operations. In contrast to use cases, a system operation only has one input interaction on

all the use cases or system operations of an interface will be listed and given a short description, and their possible order is specified (for details concerning the documentation of system interfaces in architecture reference manuals see section 3.3.3. about documenting component interfaces). The detailed behavioral specification for each use case or system operation will be given in section 3.4.

*Example*

**CellKeeper Operator Interface**

| **Interface** | CellKeeper Operator Interface |
|---|---|
| **Use Cases** | **Operator Changing Network Session**: This service allows the operator to change cell frequencies, browse cell parameter values, remove and add new cells. Changes to the network are usually not instantaneous but scheduled to happen when there is little network traffic, e.g. in the early hours of the morning. If an update fails then the network is rolled back to its previous state.<br><br>System operations of this use case: Enter_changes (operator enters changes that are transmitted to cellular network), Change_implemented (cellular network notifies CellKeeper system about successful or unsuccessful application of changes) |
| | **Network Types Updates**: This service allows the Operator to add new cell types to the network … |
| | **…** |

*Figure 5: Example CellKeeper: one of the system interfaces in an architectural overview document*

## 3.2.3 Non-Functional Requirements Section

This section documents a summary of those requirements that address aspects of the system besides functionality and that are relevant for the system architecture[5]. Non-functional requirements can be divided into three types:

1. **Qualities** – These are qualities of service (e.g. performance, throughput, usability, security, etc.) or qualities of development of the system (e.g. maintainability, supportability, portability, etc.). Often the terms used to describe qualities are subject to interpretation. The documentation of qualities here should summarize the meaning of and measures for a given quality, and leave the full details of the meaning and measures to some system requirements specification or architecture requirements specification documents.
2. **Constraints -** Constraints are conditions and limitation on the system from its environment, e.g. limitation concerning the platforms on which a system is to run (e.g. Windows/Intel compliance, fits in less than 1MB of RAM, etc.), or conditions on qualities of service (for example, average throughput can not ever be less than 100 transactions/second). Often, the constraints mentioned here are summaries of constraints specified into more details in some architecture requirements documentation.

---

the chosen abstraction level for interactions – the event or call that triggers the operation. Each input interaction triggers a new operation; any more detailed communication between actor and system is abstracted away.

The same distinction applies between component use cases and component operations.

[5] A good starting point for finding non-functional requirements is FURPS, and acronym that stands for the attributes Functionality, Usability, Reliability, Performance, and Supportability.

3. **Principles** – Principles describes the approach or strategy chosen to solve certain requirements which may be mentioned before as qualities or constraints. Principles underlay the architecture chosen. E.g., the non-functional requirement “ease of use” for a e-commerce application could be solved by having a shopping cart. Or certain development constraints could be approached by rather buying than building system components.

Non-functional requirements get referenced in subsequent sections (e.g., commentary in the overview section, mechanisms section), which explain how the chosen architecture meets these requirements.

*Example*

**CellKeeper: Qualities**

- The management system remains active even if all or part of the actual cellular network goes down. In case of partial failures of the cellular network, CellKeeper still allows operators to edit changes for the part of the network still available and can apply these changes to the network.
- Failures (e.g. power failures) of operator stations, and aborts of operator sessions (especially web sessions) should not affect other operator sessions or any other functionality of the CellKeeper system.
- Extensibility: The system must be extendable with new cell types without any downtime.
- …

**CellKeeper: Constraints**

- The system has to run on Linux.
- …

**CellKeeper: Principles**

- Interface procedures for individual cell types are stored in a repository and not hard coded, so new cell types can be added dynamically. This principle supports extensibility.
- …

*Figure 6: Example CellKeeper: non-functional requirements*

## 3.3 Structure Section

The purpose of this section is to describe the static structure of the architecture in terms of its logical components and their interconnections. The mapping of logical components to processes and code components is shown in section 3.5. Logical components are units of responsibility on a specific abstraction level. A component may correspond to a single class or a group of implementation classes[6]. Components on a high abstraction level are often called subsystems. Unless mentioned otherwise, in this document the term component is used to denote a logical component. As shown in Figure 7, the section has three parts, an overview, a description of each component, and a specification the interfaces of all components.

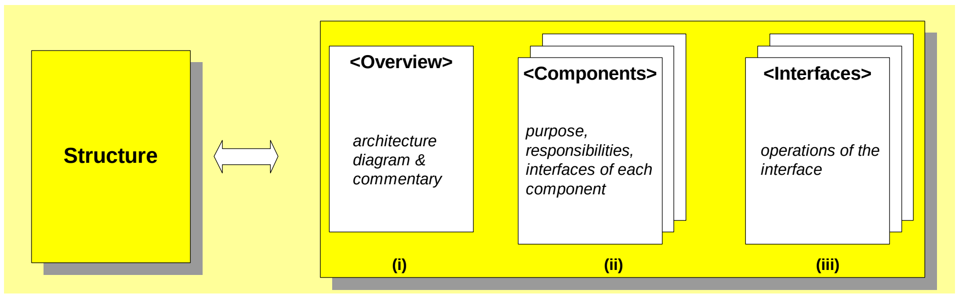

Structure
<Overview>
architecture diagram & commentary
<Components>
purpose, responsibilities, interfaces of each component
<Interfaces>
operations of the interface
(i)
(ii)
(iii)

*Figure 7: Structure section*

### 3.3.1 Overview Section

The structural overview comprises one or more *architecture diagrams* together with *commentary*. It describes the topology of the architecture.

Each **architecture diagram** shows structural elements of the system(s) and their interconnection paths. The view at the highest level of abstraction should be presented first with more detailed views following.

An architecture diagram for the logical view is conveniently expressed using the UML class diagram notation[7], in which the system is represented as a composite aggregation[8] of all its components. Components are modeled by the UML class symbol. Interconnections between components are modeled by associations. These associations can represent direct connections or they can also be used to abstract away details of more complex connection and communication patterns (e.g. indirect communication based on events, communication over a distribution system). The direction of the association shows which

[6] Components may merely serve as a high-level grouping mechanism for classes and not be reflected in the actual code (white-box component). Or components may be encapsulations of classes having an interface or façade class that is part of the component and hides the internal structure of the component (black-box component). Such interface or façade classes often have the name of the component they belong to. Components can be passive or active (have their own thread of control), be created at system startup or be created and deleted any time at runtime, be singletons or have several instances, and they can be system specific or be reusable library components or COTS. Most logical components modeled in an architecture document reflect software components, but they could also give a purely functional view of other technical components (e.g., sensors), if these are considered as part of the system and not as actors. Logical components can be mapped onto hardware nodes and into code components (code files, code libraries, code packages). That mapping can be 1:1, but this is not necessary – a logical component may be mapped to code in various packages or files, and these packages or files may contain code for various logical components.

[7] As an alternative, UML object diagrams can be used where logical components are represented as objects or object groups.

[8] Composition is a strong "whole-part" association in which lifetime of parts is co-incident with whole and if the whole is copied or deleted then so are the parts.

component initiates communication[9] on the chosen abstraction level. An association can be stereotyped to show the type of a connection[10]. Generalization and dependency relationships between components are added as needed.

Optionally, in the same or an additional diagram, the interfaces defined in section 3.3.3 can also be shown by using the round interface symbol and adding dependency arrows between the interfaces and the components using them. Although technically they are not part of the architecture, it is often convenient to include the actors that the architecture interacts with on the architecture diagram. Figure 8 and Figure 9 contain an example of architecture diagrams for the logical view.

The **commentary** describes items like the *rationale* of the architecture, outlines *architecture styles and patterns*, specifies any *architectural constraints*, and optionally also mentions *alternative architectures* not chosen (example see Figure 10).

The *rationale* behind the architecture gives the reason why a particular topology and architecture style has been chosen and why particular components exist. It relates the architecture to higher level business objectives. It explains how the architecture satisfies the architecture requirements documented in architecture requirements documents and constraints summarized in section 3.2.3. This section can also contain forward references to other sections like section 3.4.1.

*Architectural constraints* are important rules or principles governing component behavior or intercommunication, they are closely connected to the architectural styles chosen. If such a rule is violated then the architecture cannot be guaranteed to work correctly.

If *alternative architectures* have been considered, the commentary should also mention or reference these architectural concepts and give the reasons for not choosing them.

An *architectural style* defines a family of systems in terms of a pattern of structural organization. Thus it is a set of rules, which determines a set of components and the manner in which they should be connected together. Examples for architectural styles and patterns are:

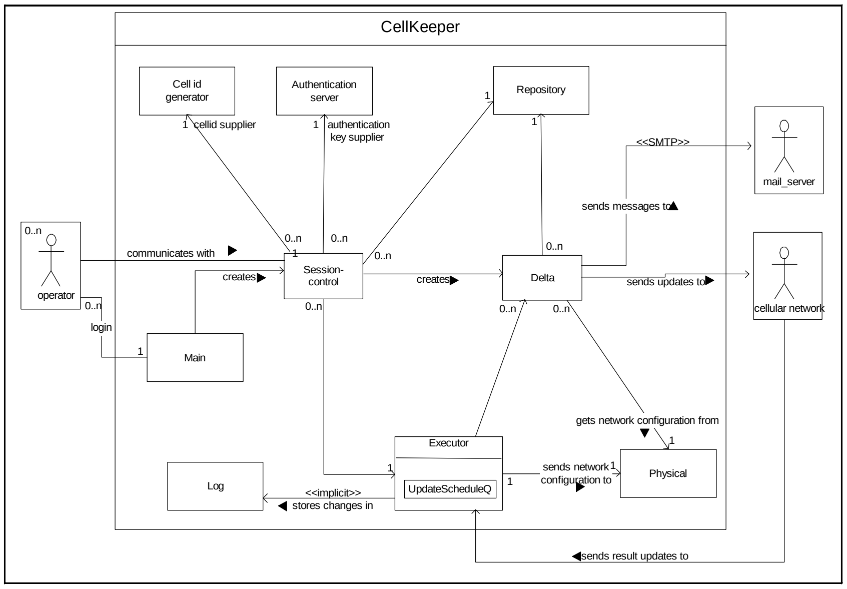
- **Layers**: System with mix of high-level and low-level abstractions and issues, where high level operations rely on lower level ones. Components are grouped into layers (subsystems) where each layer only uses operations and has associations to components from the next lower layer. Variant: associations are allowed to components from any lower layer.
- **Pipes and Filters**: System whose task is to transform a stream of data. Each processing step is encapsulated in a filter, data is passed through pipes between adjacent filters.
- **Broker, service discovery** (for distributed systems): System where it is desirable to loosely couple the client from the service provider. A broker is used for registering, discovering and connecting to services. Optionally, the broker also takes care of forwarding and marshalling all communication between client and server.
- **Model-View-Controller**: System that needs to allow the user interface to change without affecting the underlying data. System functionality and data is separated into user interface (view and control) and model (business data and functionality).
- **Event based communication**: Systems where certain components communicate with each other via implicit invocation instead of direct procedure calls. Components can subscribe to events published by other components. Variants: pushing or pulling events, event channels.

More information about architectural styles can be found in [5] and [6].

[9] Actual data flow can take place in both direction, independent of the association direction.

[10] The type of connection can refer to the underlying distribution system or communication mechanism used (e.g., CORBA, RMI, proprietary communication protocol, event broadcast, message passing). The type of connection can also refer to any higher level domain specific communication protocol used for the communication between the two components (e.g., SMTP protocol, http, or proprietary high-level protocols). The type of a connection can also depict indirect (event-based) invocation, e.g., by using the stereotype <<implicit>>. The type of the connection is given as a stereotype of the association. As an alternative, special association symbols can be defined and used.

***Example***



The CellKeeper system has one central repository for the cell management interfaces, one cell id generator, one authentication server and one executor. The system also contains an up-to-date model of the actual cellular network in the component physical. For each session with an operator, a Session_control component is created. The component Log records all changes (failed and successful ones) that have been applied to the network.

Cell Management Interfaces have to be located via a repository – so they can be changed at runtime. The interfaces have to be used indirectly through the Session-control component.

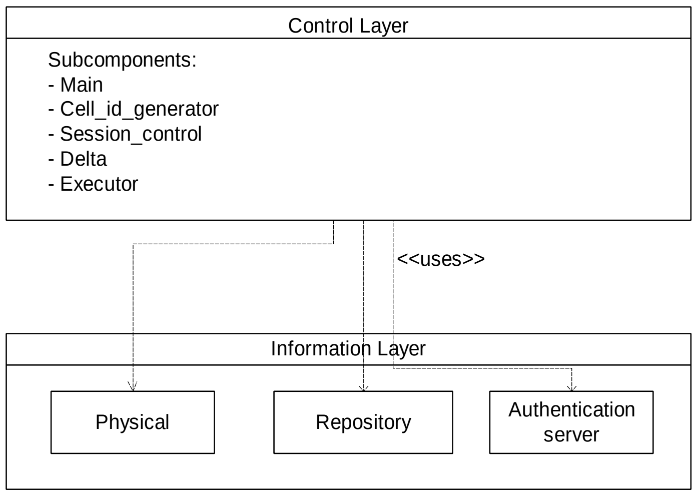
A Session_control component creates a Delta component for a set of update commands. At the end of the updates, Session_control passes a reference to Delta to the executor component and no longer uses the reference. The Executor component uses the reference to run the updates against the cellular_network.

The Physical component must be kept consistent with the actual state of the network. It is updated whenever a delta has been successfully applied to the network – but not otherwise.

The system allows several operators to edit changes in parallel. The actual execution of the changes is done sequentially. Changes are checked for consistency by the system while they are edited and before they are applied. Finally the cellular network itself can refuse changes, in which case all the changes belonging together are rolled back.

*Figure 8: Example CellKeeper: overview architecture diagram*

*Example*



This diagram shows the architecture of CellKeeper from another point of view: the layering of the various components. For the associations between the individual components see the main architecture diagram.

*Figure 9: Example CellKeeper: Architecture diagram for the layering of the logical components*

*Example*

**Commentary:**

Reasons for having Session_control and Delta components:

- For each operator session a Session_control component is created. This makes it easier to deal with aborted sessions and to guarantee that aborted sessions do not influence other sessions.
- Because a set of changes has to be dealt with together when being scheduled and applied to the network, each set of changes is being grouped together into a Delta component.

Layering of components:

- A layered architecture style has been chosen (see layering diagram). The components of the control layer access and update the information stored in the components of the information layer.
- It is conceivable that other applications than the CellKeeper would access the information of the information layer.
- The three information layer components run in separate processes, and can be deployed onto different servers, thus allowing the information components to stay alive even if the control layer crashes (see also Other views section).

Alternative architectures:
This architecture has been taken over from the prototype PfMCM. No alternative architectures have been explored due to time and cost constraints.

*Figure 10: Example CellKeeper: architecture overview commentary*

In case of large complex systems, diagrams on *different levels of abstraction* are used, with the top-level diagram showing the decomposition of the system into a handful (5 ± 2) of pieces. These pieces are high-level components; however, more often they are referred to as subsystems. Subsystems are also modeled as classes or objects in a class or object diagram. Lower level diagrams show the decomposition of the high-level subsystems or components into the components that are described in the components section 3.3.2..

If any architecture style or pattern has been used as an organizing principle (e.g., layering), a high-level diagram is used to reflect the organizing principle, resulting in the most abstract description of a system or a family of systems.

In case of a family of systems, e.g., product lines, a distinction has to be made between the generic architecture[11] of the product family and the specific architectures of individual products. Ideally, the generic architecture and the specific architectures are described in different documents that reference each other.

## 3.3.2 Components Section

This section describes each component in the architecture. A component is documented using the following template:

| | |
|---|---|
| **Component** | *A unique identifier for the component.* |
| **Responsibilities** | *Responsibilities, provided interfaces, and rationale.* |
| **Collaborators** | *Other components that the component interacts with.* |
| **Notes** | *Information about multiplicity, concurrency, persistency, parameterization, etc.* |
| **Issues** | *List of issues that remain to be resolved.* |

*Table 1: Component specification*

Figure 11 contains an example of a component specification. The following sections discuss the form and semantics of each field of the specification.

***Component***

Each component should have a unique name and possibly a version number. This section can also contain references to the component design documentation and to the implementation. In case of a complex component that has been broken down into subcomponents, also add the reference to the chapters or to the architecture document describing the internal structure of the component.

***Responsibilities***

Describes the purpose or job description of the component in terms of

1. the component's responsibilities,
2. the interface(s) that it provides.

A *responsibility* is a "piece of functionality" and specifies the purpose of the component. It can be information that the component has to know, or an action that it knows how to perform.

*Interfaces* can be listed by reference to named interface specifications in section 3.3.3, or by explicitly listing and describing the use cases or operations that constitute the interface in the same way as they are described in section 3.3.3. Referencing named interfaces facilitates component "plug and play".

It can also be useful to document the rationale for the component, i.e. give the underlying reasons why the component is designed the way it is.

[11] For modeling placeholders for architecture specific components in a generic architecture model the UML symbol for abstract classes can be used. As an alternative, placeholders in generic architectures could be modeled by a special placeholder symbol (e.g., class symbol with stereotype <<placeholder>>, operation of stereotype <<placeholder>>, class symbol with gray edges).

***Collaborators***

Lists the other components from which the component requests services in order to achieve its purpose. Besides the collaborators it is often useful to list also the specific use cases, operations or interfaces of the collaborators that are used.

***Notes***

Documents the architectural or system-level constraints on the component that the design of the component must satisfy and contains further information needed to use and understand the component. The notes section includes issues such as:

- *Multiplicity*: How many instances of this component exist in the architecture? Are the instances created and destroyed dynamically? If so, under what conditions and how does creation and destruction occur?
- *Concurrency*: Is the component multi-threaded? Does it contain data that needs to be protected against simultaneous reading and writing?
- *Persistency:* Does the component or its data need to exist beyond the lifetime of the system?
- *Parameterization:* Describes the variability of a component, i.e., ways in which a component can be parameterized or configured. This is especially important when a different configuration of the component changes the depiction of the architecture[12].

***Issues***

List of issues awaiting resolution. There may also be some notes on possible implementation strategies or impact on other components. As the understanding of the system and its architecture increases this list should eventually become empty.

[12] For example, in printer software a component might be configured for monochrome and for color, and the two configured components have different sets of collaborators or collaborations. Yet parameterizations that are concerned only with the internals (algorithms) of a component without influence on the architecture need not be documented here.

**Example**

| | |
|---|---|
| **Component** | Delta |
| **Responsibilities** | Records a history of the change commands that are requested by the operator. |
| | Knows which operator issued the commands. |
| | Incrementally develops a model of the effect of the changes on the actual network. |
| | Notifies the Operator of the effect of the changes once they have been applied to the actual network. |
| | The provided interfaces are: |
| | (i) Cell_access |
| | Provides operations for accessing and updating a model of the cellular network. See interface specification. The change commands are not directly applied to the real network, but are first recorded for later application. |
| | (ii) Update_report |
| | A single operation change_finished which is used to report the effect of the changes after they have been applied to the actual cellular_network. |
| | Note that update_report interface is necessary because only Delta knows who created the commands. |
| **Collaborators** | **interface** Cell_ access **component** Physical |
| | **interface** Cell_type **component** Repository |
| **Notes** | The component is dynamically created and destroyed. It is created whenever an Operator starts a new set of update commands. It is destroyed once these commands have been applied to the network. |
| **Issues** | How are the update commands to be recorded? |

*Figure 11: Example CellKeeper: specification of a component*

### 3.3.3 Interfaces Section

Interfaces group services (operations, use cases[4]) and allow to describe services independent of the components or system providing them. This is especially convenient if a set of use cases or operations is supported by more than one kind of component so that the set can be referred to in the description of components[13]. It is also very helpful when a component has several responsibilities, each responsibility being expressed by a different interface. Each interface is documented using the following template.

| | |
|---|---|
| **Interface** | *A unique identifier for the interface* |
| **Description** | *Brief description of the purpose of the interface.* |
| **Services**[14] | *Name and description of each use case or operation supported by the interface.* |
| **Protocol** | *Constraints on the order in which the services may be called.* |
| **Notes** | *List of components providing this interface, additional details.* |
| **Issues** | *List of issues that remain to be resolved* |

*Table 2: Interface Specification*

[13] The notion of named interfaces supported by one or multiple components occurs in distributed component technologies such as COM and CORBA, but is also commonly used in Java.

[14] With the services of an interface we mean the use cases and operations provided by this interface or by the component or system having this interface. This definition of service is not to be confused with the term e-services in e.g. e-speak where an e-service is actually a component or a whole system having one or several interfaces.

### *Interface*

Each interface should have a unique name and possibly a version number.

### *Description*

Describes the overall purpose of the interface in terms of its responsibilities.

### *Services*

Services can be requested from a component or system in order to effect some behavior. For systems they can be specified either as use cases or system operations, for components they are normally specified as operations[15]. The service section contains at least the *name* and a short *description* in terms of its responsibility for each service belonging to the interface. Depending on the level of detail and formality chosen for the documentation, services are further specified by their *signatures*, using UML, C++ or Java syntax. Further details may be added: pre- and post-conditions, possible inputs and their results, services needed from other components in order to provide the results. Yet even a detailed description should specify the services from an external point of view, and not describe their implementation or the internal structure of components providing the interface. In case the services are not invoked by a procedure call (e.g. event based systems), also specify how the services are triggered.

### *Protocol*

The protocol section contains constraints on the order in which the services may be called. For complex protocols a state machine representation (e.g. UML Statechart Diagram) should be used. For simple protocols the description can be narrative text or via the statement of pre- and post-conditions on the operations or use cases. Alternatively, sequence diagrams can be used for visualizing protocols.

### *Notes*

List of the components that provide this interface.

Operations specified in the services section of an interface specification for a component are not necessarily atomic. They may be abstractions of *conversations* where the two components actually exchange several messages or procedure calls in order to communicate the input and output parameters of the operation signature[15]. This is typically the case for user interfaces and for communications between distributed components. If operations are not atomic, the notes section should say so, reference the name of the conversation if different from the operation name, and give the section (e.g., a mechanisms section, or another interface section describing the conversation protocol), appendix or separate document (e.g., the architecture specification of this component) in which the conversation abstracted by the operation is documented. Such documentation should include the lower-level messages or operations involved as well as the constraints on their order (their protocol).

---

[15] On a lower abstraction level non atomic operations become use cases of the component, having several input and output interactions with the invoking component, which becomes an actor of the component offering the operation. An example of a non- atomic operation could be the operation "send_email(to, cc, from, text)" of an email server. While modeled as one operation in the architecture diagram, the actual communication could use SMTP, comprising operations like "HELLO(hostname):status", "RECIPIENT(…)", "DATA(…)", etc.

***Issues***

List of issues awaiting resolution.

*Example*

| | |
|---|---|
| ***Interface*** | Cell_access |
| ***Description*** | Allows access and modification of network cells. The network structure can also be explored and also be modified. |
| ***Operations*** | **Operation** *browse_cells_in_sector(sector id, list)*.<br>**Description** produces list of cell_ids and their locations belonging to the sector<br><br>**Operation** *neighbors_of(cell_id,list)*<br>**Description** lists all the neighboring cells<br><br>**Operation** *access_cell_access_proc(cell_id, "command",response)*<br>**Description** Accesses management interface of cell and returns a reference to management procedure corresponding to the command.<br><br>**Operation** *remove_cell(target_id)*<br>**Description** removes target_id cell<br><br>**Operation** *create_ new_cell(type, location, cell_id)*<br>**Description** New cell created at location. System returns identifier of new_cell_id .<br><br>**Operation** *apply(proc, c,args,resp)*<br>**Description** Applies the management interface procedure, proc, to the cell c with arguments args. The response is returned in resp.<br><br>etc. |
| ***Protocol*** | no restriction on order of operations |
| ***Notes*** | This interface may be provided by an actual cellular network or a network model. Supported by: Delta, Physical. |
| ***Issues*** | |

*Figure 12: Example CellKeeper: specification of an interface*

In communication centric systems the interfaces section may be used to specify *the conversation protocol* between two components instead of the interface of just one component, thus combining two interface specifications into one. The operations section will contain the messages exchanged, the protocol section the constraints on the order of the messages (pre- and post conditions on the messages, or sequence diagrams with pseudo-code commentary), and the notes specify the components involved in the protocol and the roles they play.

## 3.4 Dynamic Behavior Section

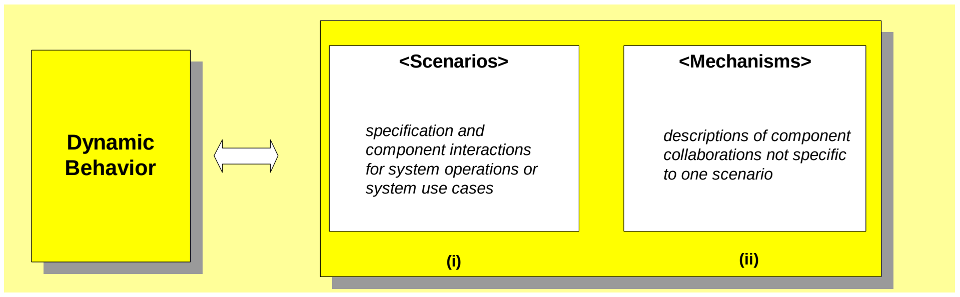

Dynamic Behavior
<Scenarios>
specification and component interactions for system operations or system use cases
(i)
<Mechanisms>
descriptions of component collaborations not specific to one scenario
(ii)

*Figure 13: Dynamic behavior section*

The dynamic behavior section specifies the behavior of the system into more details. The **scenario section** focuses on a detailed specification of the system's operations or use cases from an external as well as internal point of view. The **mechanisms section** provides important models of the systems internal behavior not covered by the scenarios section.

### 3.4.1 Scenarios Section

The scenarios section documents the dynamic behavior of the architecture for the various system use cases or system operations by specifying how the architecture responds in a number of different scenarios in response to external stimuli.

Each scenario is documented by

- a **specification** that defines how the architecture should behave in response to one or more stimuli from its environment, and
- a **component interaction model** which describes how the components collaborate to produce the behavior specified by the scenario.

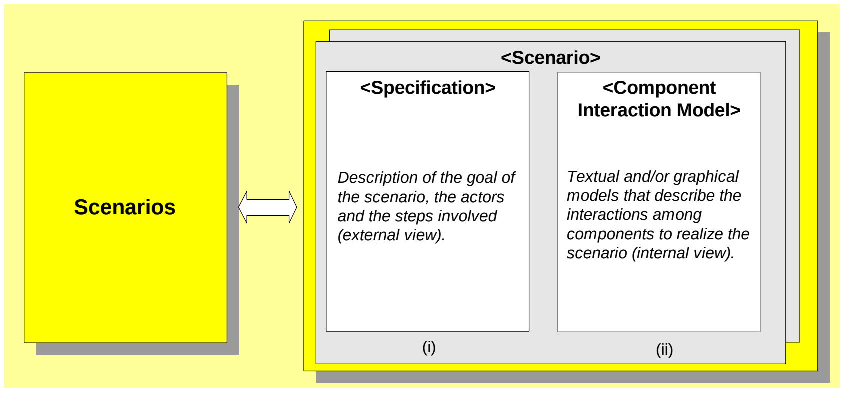

Scenarios
<Scenario>
<Specification>
Description of the goal of the scenario, the actors and the steps involved (external view).
(i)
<Component Interaction Model>
Textual and/or graphical models that describe the interactions among components to realize the scenario (internal view).
(ii)

*Figure 14: Scenarios section*

System behavior can be modeled with use cases and/or system operations. A use case consists of several steps and interactions between one or several external actors and the system in order to achieve a specific goal. If on a given abstraction level the steps of a use case are no further decomposed, the steps are considered as being system operations. In contrast to use cases, a system operation only has one input interaction on the chosen abstraction level for interactions – the event or call that triggers the operation. Each input interaction triggers a new operation. Any more detailed communication between actor and system is abstracted away. Use cases can be used for the scenario specification as well as for the interaction models. Or both, specification and interaction model can be modeled based on system operations. It is also possible to describe use cases in the scenario specification, add to the specification how the use case is divided up into system operations, and to model system operations in the interaction models.

### *Scenario Specification*

Each scenario describes a single purposeful set of interactions between some actors and the system. The specification should describe the expected behavior of the system from an *external viewpoint*. A scenario specification should avoid discussing the internal workings of the architecture. The choices for specifying a scenario are:

- A **use case**[16] **description** showing the interactions between the system and the actors in its environment – but without mentioning the behavior of components (example see Figure 15).
- A **system operation description** specifying the interactions with the system for one system operation (Figure 16 shows the system operation change_implemented; together with the system operation enter_changes it would cover the same functionality as described by the use case description in Figure 15).
- In an architectural reference manual, additionally **pre- and post-condition specifications** can be used (example see Figure 16).

The system interface section in 3.2.2 gives a high-level description of the system interfaces, it only contains a list of use cases or system operations. The scenario specification section describes them into more detail.

When use case descriptions are used in the scenario specification, the scenario specification can optionally list the individual system operations being part of the use case. As a consequence the component interaction model may be given by an interaction model for each system operation.

### *Component Interaction Model*

Most scenario specifications are accompanied by an interaction model that describes how the components collaborate in order to realize the goal of the scenario. The essential difference between the scenario specification and the interaction model is that the former has an external viewpoint while the latter has an internal viewpoint.

There are a number of choices for documenting component interactions. Commonly used techniques are use case descriptions that also contain the necessary interactions among components (as in Figure 15), one or more sequence diagrams showing exemplar interactions between components and actors (as in Figure 17), UML activity diagrams, or collaboration or sequence diagrams accompanied by a pseudo-code specification (as in Figure 16). Of course, in contrast to scenario specifications, component interactions must discuss how components collaborate!

Typically there are a handful of key scenarios, which must be understood in order to understand the system as a whole; these are the ones that should be documented in an architectural overview document. The overview should also capture any mechanisms that deal with high-risk issues, e.g. real-time performance or security. In contrast to an architectural overview document, a reference manual should document all the use cases or system operations that are identified during the architectural design process. In most cases the modeling techniques are used in a less formal way for architectural overview documents, whereas more emphasis is put on formality, details and completeness in architectural reference manuals, even if the same modeling techniques (e.g. sequence diagrams or use case descriptions) are used.

---

[16] Often use cases are more appropriate for interactions with human actors as they allow to model a sequence of interactions with the human actor in one use case, and operations are more appropriate for services requested by software actors (other systems), as they typically have only one input event and only return one result event to the requesting system. Another criterion for choosing the abstraction level of use cases or of system operations for modeling the system behavior is the complexity of the individual system operations: if they are trivial, rather model on the level of the use case the operations are part of in order to minimize the number of scenario specifications and interaction models needed.

Interactions that occur in various scenarios can be modeled in the mechanisms section 3.4.2 and can be referenced in the interaction models provided here.

*Example*

### Scenario Specification: Operator Session for Changing Network

The architecture has to be able to deal with updates to the cellular network. Updates may be entered simultaneously, updates may be inconsistent and multiple updates may be scheduled for the same time.

| | |
|---|---|
| **Use case** | OperatorChangingNetworkSession |
| **Description** | Operator session for using system to manage network. |
| **Actors** | Operator (primary)<br>Cellular_network |
| **Steps** | 1. Operator logs in *(user_id).* System allows only valid operators to proceed.<br>2. Operator performs any number of changes to cells. Each change is scheduled to happen at a later time – usually in the early hours of the morning after the operator has logged out.<br>3. After the changes have been completed the system notifies the operator of the results of the changes to the actual network; if a change fails then the network is rolled back (see scenario Change_implemented). |

### Component Interaction Model for OperatorChangingNetworkSession

| | |
|---|---|
| **Use Case** | OperatorChangingNetworkSession |
| **Description** | Operator session for updating the network. |
| **Actors** | Operator, Cellular_network |
| **Components** | :Session_Control, :Delta, :Physical, :Executor |
| **Assumptions** | 1. Most updates are independent of each other.<br>2. The consistency check will catch most updates that are going to fail if applied to the network.<br>3. :Executor ensures that only one set of changes to the cellular network are in progress at any one time. |
| **Steps** | 1. A :Session_Control component is created when an operator logs in. For each set of updates that the Operator works on a :Delta component is created to record the changes. As the Operator accesses cells :Delta accesses :Physical to locate cell parameter values.<br>2. When the Operator has finished a set of changes :Delta checks whether they are consistent with the state of :Physical. If they are, the :Delta component is sent to the :Executor component to be activated at the appropriate time.<br>3. When that time arrives the changes are again checked for consistency with the (current) state of :Physical. If they are consistent then the changes are applied to the cellular_network.<br>4. When the cellular network has completed the changes it informs the :Executor whether they were successful. If the changes failed then the :executor rolls the cellular network back to its original state.<br>5. If successful then the :Physical component is updated to bring it in line with the cellular network.<br>*6.* In both cases the :Operator is emailed of the result of the changes. |
| **Variations** | **#6.** If the operator is logged in then the notification will also be sent to the Operator's terminal. |

*Figure 15: Example CellKeeper: high-level specification and textual component interaction model for the use case "OperatorChangingNetworkSession"*

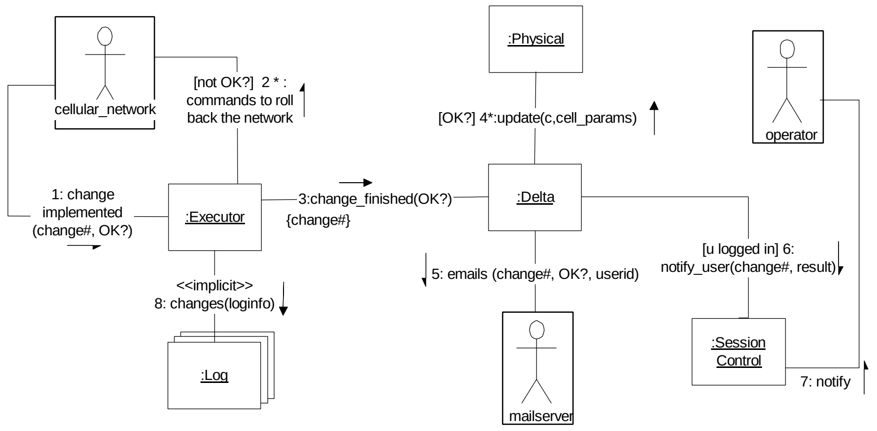
## *Example*

### Scenario Specification: Network issues Change_implemented

| | |
|---|---|
| **System Operation** | Change_implemented(change#, OK?) |
| **Description** | The operation change_implemented is invoked by the network when it has completed an update. |
| **Actors** | Cellular_network (primary)<br>Operator, Mail_server |
| **Input** | change#: reference number of the changes implemented<br>OK?: reports whether changes have been successful |
| **Output** | rollback_to_pre {Cellular_network}; notify_user {Operator}; email {Mail_server} |
| **Preconditions** | - |
| **Postconditions** | IF OK? THEN updated cells in change# have been copied to :Physical<br>ELSE rollback_to_pre (change#) has been sent to cellular_network.<br>email (change#, ok?, userid) has been sent to mailserver.<br>IF operator issuing the changes is still logged in, notify_user(change#, ok?) has been sent to operator. |

### Component Interaction Model: Network issues "Change_implemented"

using collaboration diagram with comment for change_implemented(change#, OK?)


cellular_network
[not OK?] 2 * : commands to roll back the network
:Physical
[OK?] 4*:update(c,cell_params)
operator
1: change implemented (change#, OK?)
:Executor
3:change_finished(OK?) {change#}
:Delta
[u logged in] 6: notify_user(change#, result)
5: emails (change#, OK?, userid)
<<implicit>> 8: changes(loginfo)
:Log
mailserver
:Session Control
7: notify

1. Network notifies :Executor about applied changes.
2. If the change to the network fails (i.e. not OK?) the :Executor issues the network commands to rollback the actual network to its previous state.
3. The :Executor also notifies the :Delta component that had initiated the network update of the result of the change.
4. If the change was successful then :Delta updates :Physical, i.e. the internal network model.
5. :Delta also emails the result of the change to the Operator.

6,7. If the Operator is still logged in, :Delta sends a message to the :Session Control component so that the Operator can be notified online.

8. All failed and sucessful changes are sent to any :Log components having subscribed to receive that information.

*Figure 16: Example CellKeeper: detailed specification using pre- and post-conditions and component interaction model using a collaboration diagram with pseudo-code specification for the system operation "Change_implemented"*

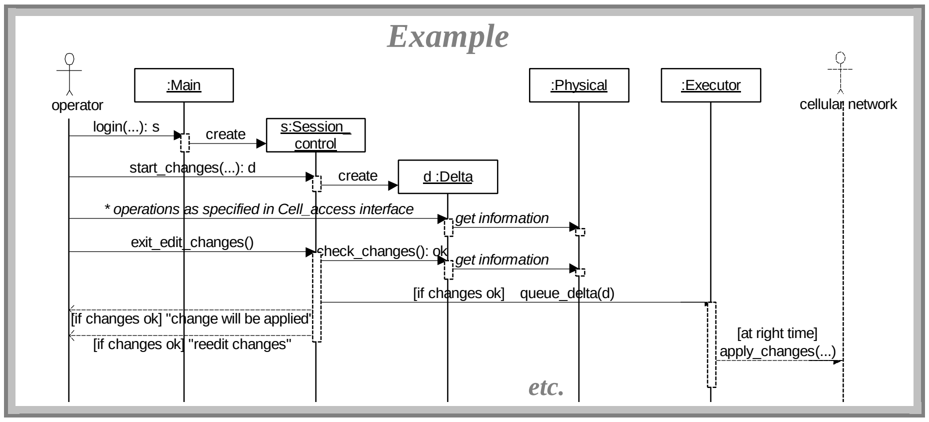


*Figure 17: Example CellKeeper: an alternative component interaction model for the use case "OperatorChangingNetworkSession" specified in Figure 15*

### 3.4.2 Mechanisms Section

The mechanisms section allows the explanation and documentation of issues concerning system behavior and component collaboration that have been postponed or abstracted away in previous chapters. The mechanisms section:

- explains the component collaborations provided for satisfying non-functional quality and constraints requirements listed in section 3.2.3 that concern the behavior of the system[17] (e.g., in distributed systems how data replication works and how data consistency is guaranteed, how performance improvements are achieved by caching data locally and how data integrity is cared for, or for systems with security concerns how authentication, authorization and accounting works),
- gives details to architecture styles and patterns used (e.g., it provides more detailed architecture diagrams to explain the mechanisms used for implicit invocation as in Figure 18 or to explain the mechanisms used to obtain a reflective architecture),
- describes how use cases work together,
- describes repeating interaction patterns that are part of several system use cases or system operations and therefore their description has been deferred to the mechanisms section.

The mechanisms section is optional; if all behavioral issues have been modeled and explained into enough details in previous sections, the section is omitted.

The mechanisms section contains text, class diagrams, objects diagrams, and interaction diagrams showing specific collaborations.

[17] The satisfaction of all the non-functional requirements is assessed in the conclusion section 3.6. Here the focus is only on modeling important mechanisms that help to fulfill requirements concerning system behavior.

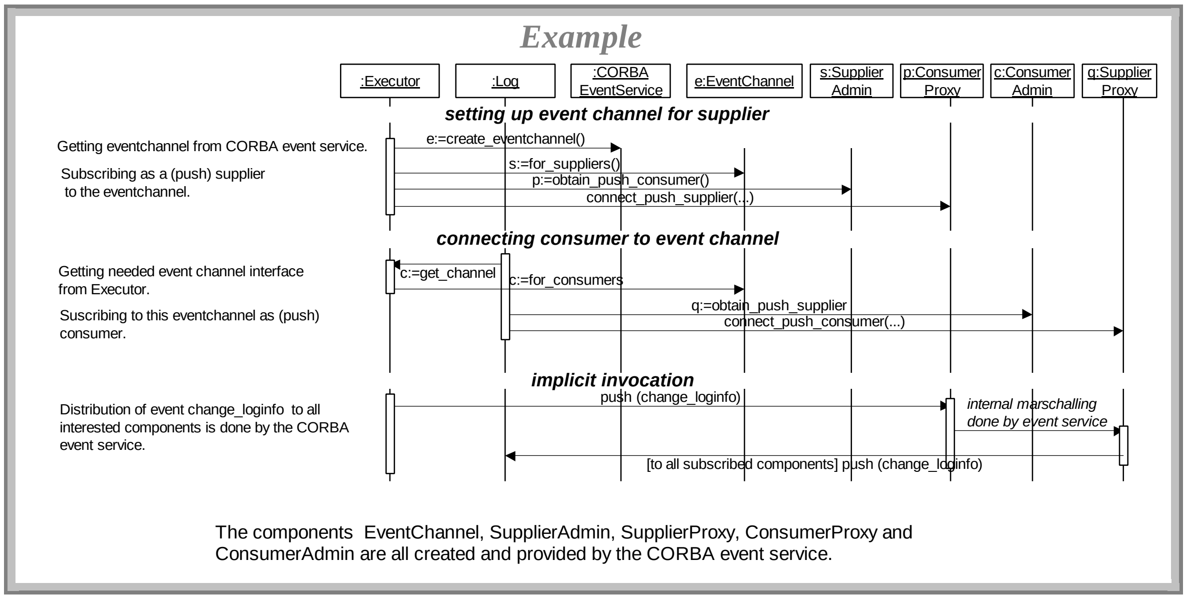

Example
:Executor
:Log
:CORBA EventService
e:EventChannel
s:Supplier Admin
p:Consumer Proxy
c:Consumer Admin
q:Supplier Proxy
setting up event channel for supplier
Getting eventchannel from CORBA event service.
Subscribing as a (push) supplier to the eventchannel.
e:=create_eventchannel()
s:=for_suppliers()
p:=obtain_push_consumer()
connect_push_supplier(...)
connecting consumer to event channel
Getting needed event channel interface from Executor.
Suscribing to this eventchannel as (push) consumer.
c:=get_channel
c:=for_consumers
q:=obtain_push_supplier
connect_push_consumer(...)
implicit invocation
Distribution of event change_loginfo to all interested components is done by the CORBA event service.
push (change_loginfo)
internal marschalling done by event service
[to all subscribed components] push (change_loginfo)
The components EventChannel, SupplierAdmin, SupplierProxy, ConsumerProxy and ConsumerAdmin are all created and provided by the CORBA event service.

*Figure 18: Example CellKeeper: mechanism for the implicit invocation of methods of the Log component*

## 3.5 Other Views Section

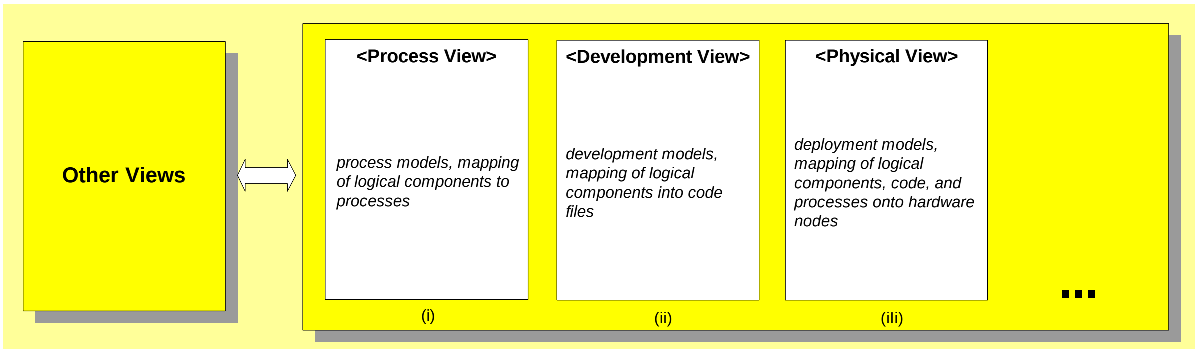

Other Views
<Process View>
process models, mapping of logical components to processes
(i)
<Development View>
development models, mapping of logical components into code files
(ii)
<Physical View>
deployment models, mapping of logical components, code, and processes onto hardware nodes
(iii)
...

*Figure 19: Other views section*

The other views section describes the process, development and physical view of the architecture, and shows how the logical components map into these other views. The sections specify:

- the static model (the structure and interconnections) of the processes, code components or hardware nodes,
- the dynamic model of the processes, code components or hardware nodes for specific scenarios (examples of interactions among the elements),
- mapping of the logical components described in sections 3.3 and 3.4 onto processes, code components or hardware nodes.

All three views are optional. Whether they are included in the architecture document or not depends on the purpose and audience of the document as defined in section 3.1.

### 3.5.1 Process View Section

The process view section depicts the process view of the system and maps the logical view into the process view. It describes the system’s decomposition into executing tasks and processes by assigning logical structural elements to threads of control, and grouping threads and processes. For the static model of processes and threads and the mapping of logical components a class or object diagram is used. Processes and threads are modeled by classes or objects of stereotype <<process>> or <<thread>>. Composition is used to model the assignment of logical elements (active and non-active components of the logical view) to processes and threads (example see Figure 20). Composition is also used to model any hierarchy between processes and threads. Processes or threads interacting and communicating with each other may be linked by associations.

Optionally sequence or collaboration diagrams may be added showing the interactions taking place between processes for specific scenarios or mechanisms specified in section 3.4.1 or 3.4.2. Of course such interaction diagrams have to be consistent with the component interaction models of section 3.4 and the mapping specified above.

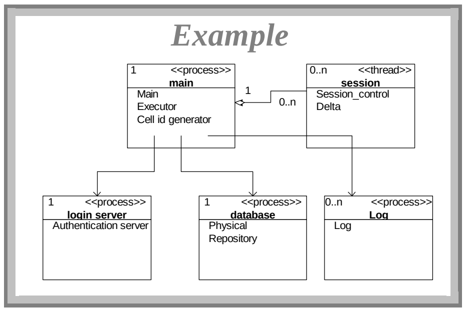


*Figure 20: Example CellKeeper: static model of process view*

### 3.5.2 Development View Section

The development view section explains the development view of the system and shows how the logical view maps into the development view. It describes the decomposition of the system's code base into code components like files and libraries, dependencies between these code components, and the assignments of logical components of the logical view to code components. The UML package symbol can be used for showing the various groups of code files and their implementation dependencies (e.g., compile or usage dependencies). The assignment of logical components to packages can be done either by listing all logical components in the description of the package, or by explicitly using the file symbol or the UML component symbol for components, and modeling in a diagram the composition of the code components into packages and the assignment of logical components to code components. The description of the code components, their interdependencies, and the assignment of logical components to code components can also be done textually.

***Example***

***Implementation mapping:***

- For each logical component a directory is created that contains all code files belonging to that component.
- For each class a separate file is created.

*Figure 21: Example CellKeeper: mapping of logical components to code components*

### 3.5.3 Physical View Section

The physical view section describes the deployment of the system. It shows the architecture of the hardware nodes and maps processes, logical components or code components onto hardware nodes. It thus allows us to show how the software is deployed onto the hardware infrastructure.

In a UML deployment diagram processes and logical components are modeled as objects, code components by the UML component symbol, and nodes by the UML symbol for nodes. Apart from the containment relationship from hardware nodes to processes, logical components or code components, the diagram can also use any other model elements of static diagrams, especially dependencies and interfaces. Associations between nodes can be added to show network connections, their names or descriptions can contain the name and type of network used. Non-UML diagrams or pictures may also be used to explain the hardware infrastructure.

Optionally sequence or collaboration diagrams may be added showing the interactions taking place between hardware nodes for specific scenarios or mechanisms specified in section 3.4.

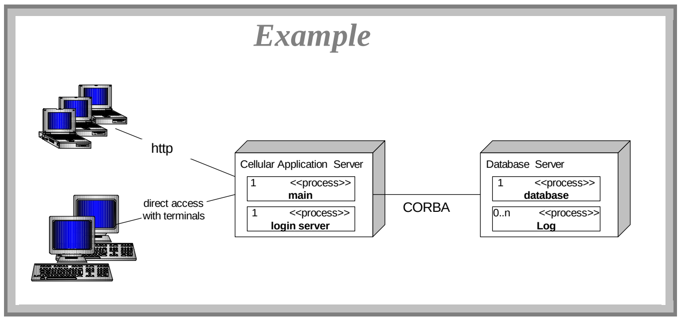


*Figure 22: Example CellKeeper: hardware infrastructure*

## 3.6 Conceptual Framework Section

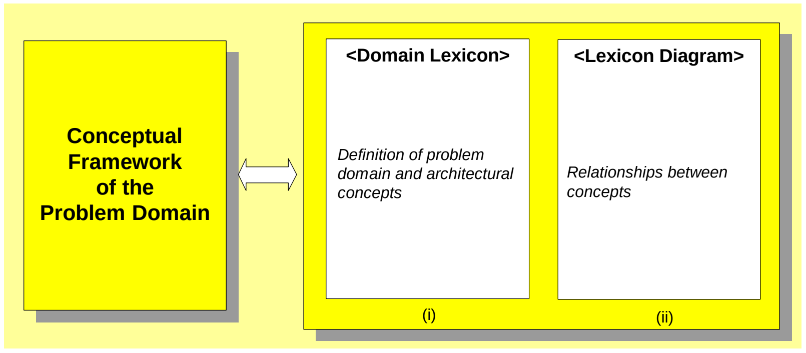

Conceptual Framework of the Problem Domain
<Domain Lexicon>
Definition of problem domain and architectural concepts
(i)
<Lexicon Diagram>
Relationships between concepts
(ii)

*Figure 23:Conceptual framework section*

The conceptual framework refers to a network of concepts and the relationships between them. These concepts are documented in a glossary-like form, which we call a domain lexicon, and visualized in a diagram, the lexicon diagram. In a **domain lexicon** each of the important terms in the problem domain and those terms used in discussions about the system and its architecture are defined. The most important aspect of the lexicon is to have the definitions of terms be as specific and unambiguous as possible. Figure 24 contains an example.

A supplemental model of the conceptual framework is the **lexicon diagram**[18]. It depicts graphically the relationships between concepts and can convey complex relationships at a glance (see Figure 25 for an example). The UML class diagram notation is used for the lexicon diagram: concepts are modeled by the class symbol (without operation or attribute compartment), relationships between concepts by generalization, aggregation and association symbols[19].

It is appropriate for the conceptual framework section to contain other diagrams or pictures explaining the terms of the problem domain (example see Figure 26). This is especially helpful if the concepts are geometric or visual in nature.

***Example***

**Cell** - A cell in a cellular telephone network handles all the calls of mobile phones in the area it covers. To do this, it transmits identifying information on a beacon radio frequency and handles the "traffic" on other frequencies. In busy areas like cities, cell diameters are measured in meters, in rural areas the diameters can be up to 64 km. The only limit is the interference of cells that reuse the same frequency.

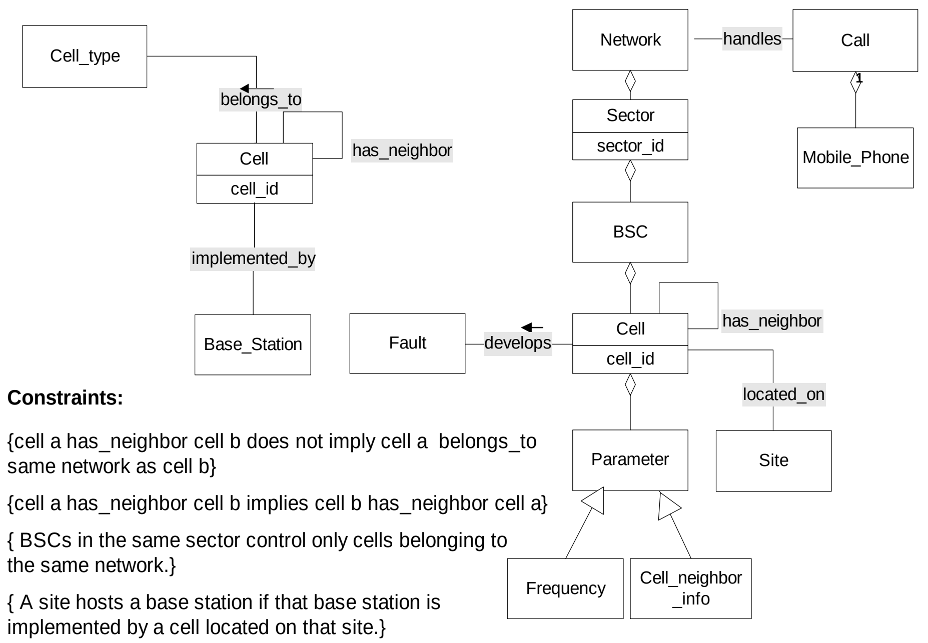
**Base station -** A base station implements a cell. A base station is a computer plus radio equipment that is connected to a telephone exchange. Base stations are located on a "site". The sites are the most visible part of a cellular network because of the very obvious antenna masts that are visible all over the country. One site can host multiple base stations. The base stations are connected to a telephone exchange that, besides switching phone calls, is also responsible for controlling the operation of the cells. In the GSM system it is called the Base Station Controller (BSC).

*Figure 24: Example CellKeeper: domain lexicon*

[18] As the lexicon diagram models the concepts of the problem domain and not the software components it is called the domain class diagram in some OOAD methodologies.

[19] Alternatives to UML class diagrams are semantic networks, concept maps (see Novak[7]), or any other graphical representation for describing ontology.

## Example



**Constraints:**

{cell a has_neighbor cell b does not imply cell a belongs_to same network as cell b}

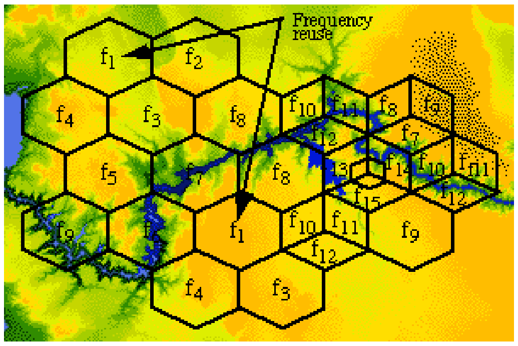
{cell a has_neighbor cell b implies cell b has_neighbor cell a}

{ BSCs in the same sector control only cells belonging to the same network.}

{ A site hosts a base station if that base station is implemented by a cell located on that site.}

*Figure 25: Example CellKeeper: lexicon diagram for the CellKeeper domain*

## Example



Above picture shows the cells of a cellular network with the frequencies assigned to the individual cells. A cell in a cellular telephone network handles all the calls of mobile phones in the area it covers. To do this, it transmits identifying information on a beacon radio frequency and handles the "traffic" on other frequencies. When a network is initially built, the area that is covered by each cell is very large. When the number of subscribers increases, capacity is added by splitting cells. This results in more and smaller cells, offering more total capacity. This technique makes cellular networks highly scalable. In busy areas like cities, cell diameters are measured in metres, in rural areas the diameters can be up to 64 km. The only limit is the interference of cells that reuse the same frequency.

*Figure 26: Example CellKeeper: picture showing a typical distribution of cells.*

## 3.7 Conclusion Section

The conclusion sections describes conclusion drawn from the architecture models and the activity of architecting. It contains assessments, as well as past and current open issues.

The **assessment section** should attempt to assess the architecture against its requirements. The assessment section:

- highlights the advantages and known limitations of the chosen solution,
- describes how well non-functional requirements are met,
- lists any known inconsistencies within the architecture document[20] (terminology, deviations from the requirements stated in section 3.1),
- optionally discusses possible directions for the evolution of the architecture,
- optionally[21] documents any consistency analysis done across the architectural views presented in sections 3.1 to 3.5.

For all non-functional quality and constraint requirements listed in section 3.2.3, the assessment section describes with which techniques and mechanisms and how well the requirement is met. For requirements that concern the behavior of the system (e.g., security, availability), the description of how the requirement is met can reference section 3.4.2.

Issues raised during the design of the architecture or during the review of documentation about the architecture should be recorded in the **open issues section.** The documentation for each issue should contain information about the origin of the issue and the tracking of it until it is resolved. Table 3 contains a sample template for issues.

| ***Id*** | ***Risk*** | ***Origin*** | ***Description*** | ***Owner*** | ***Resolution*** | ***Status*** |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |

*Table 3: Open issues section*

[20] Inconsistencies between the architecture document and other requirements documents are dealt with in section 3.1. of the architecture document.

[21] Needed for conformity with IEEE Recommended Practice for Architectural Description

## 4. Conclusion and Acknowledgments

This document proposes a template for documenting software architectures. A copy of the template as MS-Word document can be obtained by email from the authors.

Describing architectures is not a new issue. Several books have been published on architecture styles and architecture patterns, e.g., [5] and [6]. Various Architecture Description Languages (ADLs) have been developed, though these are primarily oriented towards formal descriptions of component interfaces and connectors. Also most books on UML modeling emphasize the importance of documenting the high-level architecture of a system. This architecture documentation template goes a step further in that it proposes the actual structure of the documentation and gives guidelines how and where to document the various views of an architecture. This has been possible as this template has evolved from architecture documentation work done in various projects within HP. A similar approach to the one presented in this paper can be found in [8], a standard for documenting IT architectures presented in a recent issue of the IBM Systems Journal devoted to IT systems and their architecture.

Thanks are due to the other members of the PGS architecture practice area: Dana Bredemeyer, Ron Grace, Mark Interrante, Ruth Malan, and Steve Rhodes. An important starting point has also been the SEI's Software Architecture Analysis Method [1].

Since the first release of this template it has been used on a number of projects within Hewlett-Packard on a variety of types of software: 1) embedded software in test instruments, printers and other devices, 2) device drivers, 3) graphic display of system configuration data.  In the course of our using this template to consult with teams that were documenting architecture we received and incorporated numerous comments on the template. We particularly want to thank Jonathan Patrizio and Don Marselle (HP Cupertino), Jon Lewis and Lee Jackson (HP Vancouver), Axel Wankmueller (Agilent Technologies Boeblingen), and Pat Fulghum (HP Boise).

## 5. References

[1] Bass L., Clements P. and Kazman R., Chapter 9 of *Software Architecture in Practice*, Addison-Wesley, 1997.

[2] IEEE Draft for Standard, *IEEE P1471/D5.1 Draft Recommended Practice for Architectural Description*, October 1999

[3] *CellKeeper, a cellular network manager*, http://www.bytesmiths.com/pubs/DesignFest96/cells.html

[4] Kruchten, Philippe, *The 4+1 View Model of Architecture*, IEEE Software, November, 1995.  A version of the this paper is also available on line at: http://www.rational.com/uml/resources/whitepapers/**.**

[5] Shaw, Mary and Garlan, David, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[6] Buschmann, Frank, et al., *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley & Sons, 1996.

[7] Novak, J.D. and D. B. Gowin, *Learning How to Learn*, New York: Cambridge University Press, 1984.

[8] Youngs, R., et al., *A standard for architecture description*, IBM Systems Journal, Vol 38, No. 1, 1999.

## Appendix A: Outline Summary

The following paragraph gives an overview of the typical outline of an architecture document as specified in this template:

1 **Introduction**: general information about the architecture document, other related documents

2 **System Purpose**: purpose, functional and qualitative characteristics of the system from a black box point of view

- 2.1 **Context**: description of the context of the system and the problem it solves
- 2.2 **System Interface**: services provided by the system
- 2.3 **Non-functional Requirements**: principles of the architecture, qualities of services, external constraints

**3 Structure**: logical view of the static structure of the architecture in terms of its components, their interconnections, and the interfaces and operations offered by the components

- 3.1 **Overview**: architecture diagram(s) for overall structure, commentary containing rationale, architecture style, architectural constraints and alternative architectures
- 3.2 **Components**:
  - *for each component*: description of the component in terms of its responsibilities, interfaces it offers, collaboratoring components, possibilities for parameterizations, and constraints
- 3.3 **Interfaces**:
  - *for each component interface:* description of the interface, its operations and constraints on the order of operations, optionally specification of the operations

4 **Dynamic Behavior**: specification of the system behavior, collaboration of components for achieving system behavior

- 4.1 **Scenarios**: architecture diagram(s)

  *for each scenario:*
  - specification of the scenario as system operation or use case specification for some or all of the services provided by the system
  - component interaction model of the scenario
- 4.2 **Mechanisms (optional)**: explanation of and interaction models for important mechanisms

5 **Other Views**:

- 5.1 **Process View (optional)**: architecture diagrams of the processes showing the mapping of components to processes
- 5.2 **Development View (optional)**: mapping of logical components to code components, structure of code components
- 5.3 **Physical View (optional)**: architecture diagrams of the hardware infrastructure showing the mapping of components, processes and code files to hardware nodes
- 5.x **Additional Sections (optional)**: sections for additional models and viewpoints not specified in this template

6 **Conceptual Framework**: definition of the problem domain specific concepts and terms and their relationships

7 **Conclusion**: advantages and limitations of the architecture in respect to non-functional requirements, open issues

***Internal and external view of system behavior (logical view)***

Logical system and component behavior can be modeled from an *internal or an external point of view.* Of course, the specification for the external view of the behavior has to be consistent with any model for the internal view of the behavior. Table 5 gives an overview of where which view is modeled in the architecture documentation.

| | ***System*** | ***Component*** |
|---|---|---|
| ***External view*** | • Context Section 3.2.1.<br>• System Interface Section 3.2.2.<br>• Scenario specifications in Dynamic Behavior Section 3.4. | • Components Section 3.3.2,<br>• Interfaces Section 3.3.3 |
| ***Internal view*** | • Component interaction models for scenarios in Dynamic Behavior Section 3.4. | See external system views in the architecture document for the specific component. |

*Table 4: External and Internal Behavioral Views*

## Appendix B: Conformance to the IEEE Recommendation for Architectural Description

The *IEEE Draft Recommended Practice for Architectural Description* [2] gives a conceptual framework and guidelines for documenting architectures. Table 6 shows how the template presented here conforms to the IEEE recommendation.

| *IEEE Recommendation* | *This Template* |
|---|---|
| **Section 5.1.: General information about the architecture documentation** | |
| Date of issue and status, issuing organization, change history, summary | Introduction Section 3.1. |
| Scope and context | Context Section 3.2.1. |
| Glossary | Conceptual Framework Section 3.6. |
| References | Introduction Section 3.1. |
| **Section 5.2.: Stakeholders and concerns** | |
| Detailed identification[22] of stakeholders (audience) | Introduction Section 3.1. |
| Mission of the system | System Purpose Section 3.2., especially Context Section 3.2.1 and System Interface Section 3.2.2. |
| Concerns like maintainability, deployability, and evolvability as well as other concerns of stakeholders | Non-functional Requirements Section 3.2.3., Introduction Section 3.1. |
| Appropriateness, feasibility, and risks | Assessment and Issue Sections of the Conclusion Section 3.7. |
| **Section 5.3.: Selection of architectural viewpoints** | |
| Selection and customization of viewpoints | Introduction Section 3.1. |
| Library viewpoints specified by template | see table 6 |
| **Section 5.3.: Architectural views** | Conceptual Framework Section 3.6., System Purpose Section 3.2., Structure Section 3.3., Dynamic Behavior Section 3.4., Other Views Section 3.5. |
| **Section 5.5 and 5.6: View consistency and architectural rationale** | |
| Known inconsistencies among views, consistency analysis | Conclusion Section 3.6. |
| Consistency rules | see below |
| Rationale for selected architectural concepts | Commentary in Overview Section 3.3.1. |
| Alternative architectural concepts | Commentary in Overview Section 3.3.1. |

*Table 5: Mapping of IEEE Recommendation to Architecture Documentation Template*

**Viewpoint specifications** are required by the IEEE Recommendation for Architectural Description [2]. They have to be referenced in the introduction section of an architecture document in order for the document to be conformant to [2]. Stakeholders addressed in the viewpoints depend on the type of architecture document (overview or reference manual) to be created (see chapter 2 of this paper). Stakeholders and their concerns can be further detailed in the introduction section of the architecture document.

[22] The template can be used to create an *architectural overview* for a wide range of stakeholders – this is typically done in the early phases of the development. Or the template can be used to create an *architectural reference manual,* which is a living document mainly for system developers and guides the system developers throughout the life cycle of the system even in maintenance and enhancements. The template can be used for the system as a whole and/or for specific components of the system.

The template focuses on the description of software and firmware systems. The following viewpoints are part of the template: Context and System Interface Viewpoint, Concepts Viewpoint, Logical Structure Viewpoint, Logical Behavior Viewpoint, Process Viewpoint (optional), Physical Viewpoint (optional), Development Viewpoint (opional). Table 7 contains a specification summary of the viewpoints. The section titles refer to both, to the more detailed specification of the viewpoints in this architecture template, and to the sections containing the appropriate views in the architecture documentation

| *Name of viewpoint* | *Concerns addressed* | *Models etc.* | *Sections* |
|---|---|---|---|
| **Context and System Interface Viewpoint** | What is the boundary of the system? What other entities does the system interact with? What are the responsibilities of the system? What are the main services offered by the system? | - Context diagram (UML use case diagram, UML class diagram, data flow diagram)<br>- System interface description | System Purpose Section 3.2. |
| **Concepts Viewpoint** | What are the important terms of the problem domain? How are the terms and concepts used in the architecture description defined? How do the different concepts relate to each other? | - Domain lexicon<br>- Lexicon diagram | Conceptual Framework Section 3.5. |
| **Logical Structure Viewpoint** | What is the static structure of the system? What are the components of the system? What are their interfaces? How do they interconnect? What is the rationale for the chosen structure? | - Architecture diagram (UML class diagrams)<br>- Commentary<br>- Component specification<br>- Interface specification | Structure Section 3.3. |
| **Logical Behavior Viewpoint** | How does the system respond to stimuli from the environment?<br>How do the components interact to produce the desired behavior? | - Scenario specification<br>- component interaction model (UML collaboration or sequence diagram, text) | Dynamic Behavior Section 3.4. |
| **Process Viewpoint** | How are the components of the systems assigned to threads of control? | - UML object model with active objects | Process View 3.5.1. |
| **Physical Viewpoint** | How are the components allocated to physical nodes? | - UML deployment diagram | Physical View 3.5.3. |
| **Development Viewpoint** | How is the code structured and how are logical components assigned to code pieces (files, ...)? | - UML component diagram<br>- text | Development View 3.5.2. |

*Table 6: Viewpoint Specifications in Architecture Documentation Template*

The following **consistency rules** apply between the different views:

- The *components* (modeled as UML objects) in the component interaction models of the logical behavior view, in the process view, and in the physical view must be specified in the logical structure view.
- The *interactions* in the component interaction models of the logical behavior view correspond to component operations in the logical structure view.

The *use cases* and *system operations* specified by scenario specifications in the logical behavior view correspond to the use cases and system operations listed in the system interface section of the context and system interface view.

Additional viewpoints can be specified and modeled in an architecture document. These viewpoints can either be part of one of the above viewpoints (e.g., special viewpoints part of the logical structure viewpoint, e.g. a viewpoint addressing security), or they can show the system architecture from a totally different angle and thus contain elements from various of the above viewpoints or from various subsystems or components. Examples for additional viewpoints are: data view (ER-model of data stored in a relational database, extension to the logical structure view), database view (showing all components and behavior relevant to storing data), network view or communication view, infrastructure view (containing only basic services), application view, etc. Whether and which additional viewpoints are relevant is system specific. Additional viewpoints are specified in section 3.1, and are modeled in additional sections 3.x.

## Appendix C: Glossary

| | |
|---|---|
| **active object** | An object that owns a thread and can initiate control activity. |
| **actor** | An active entity (human user or external system) that is in the environment of the system and that interacts with the system. An actor represents a coherent set of roles – one user can perform several roles and several users can play the same role. |
| **architecture** | The fundamental organization of a software or firmware system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. |
| **architectural documentation template** | A document specifying the structure and content of an architectural documentation. |
| **architectural overview document** | An architectural document giving an overview of the architecture at a high level of abstraction. The document is targeted at a broad range of audiences including developers, marketing, management and possibly potential end-users. |
| **architectural reference manual** | An architectural document specifying an architecture in a detailed and precise manner. The document is targeted at developers and should be updates as changes occur so it always reflects the actual architecture. |
| **architectural style** | Defines a family of systems in terms of a pattern of structural organization. Thus it is a set of rules, which determines a set of components and the manner in which they should be connected together. A style or pattern describes a generic solution to a specific class of problems appearing typically in a specific context. |
| **architecture pattern** | see *architectural style* |
| **architecture** | see *software architecture* |
| **class diagram** | A diagram that shows declarative model elements (e.g., logical components, interfaces, classes) and their contents and relationships. |
| **code component** | A package or a set of files or directories that contain the implementation of a piece of a software system, including software code or equivalents such as scripts or command files. |
| **collaboration diagram** | An interaction diagram that shows the components and their relationships and the messages exchanged between the components. A collaboration diagram is a special case of an object diagram. |
| **collaborators** | Components that interact with another component. |
| **communication** | see *conversation* |
| **component** | A unit of responsibility and functionality on a specific abstraction level. A component may correspond to a single class or a group of implementation classes. Components may merely serve as a high-level grouping mechanism for classes and not be reflected in the actual code (white-box component). Or components may be encapsulations of classes having an interface or façade class that is part of the component and hides the internal structure of the component (black-box component). Such interface or façade classes often have the name of the component they belong to. Components can be passive or active (have their own thread of control), be created at system startup or be created and deleted any time at runtime, be singletons or have several instances, and they can be system specific or be reusable library |

| | |
|---|---|
| | components or COTS. Most logical components modeled in an architecture document reflect software components, but they could also give a purely functional view of other technical components (e.g., sensors), if these are considered as part of the system and not as actors. Logical components can be mapped onto hardware nodes and into code components. That mapping can be 1:1, but this is not necessary – a logical component may be mapped to code in various packages or files, and these packages or files may contain code for various logical components. Components on a high abstraction level are often called subsystems. |
| **composition** | An association that specifies a whole-part relationship between the aggregate (whole) and its parts. The parts are life-time dependant of their aggregate, i.e., they cannot exist before or after the aggregate exists. |
| **context diagram** | A diagram that shows a system or component and the actors it interacts with. |
| **conversation** | An exchange of messages between two components for a specific purpose (e.g., the information needed to start an operation is transferred in several messages). A conversation is often modeled as one interaction or message on a higher abstraction level, and then split up into a sequence of messages on a lower abstraction level. The conversation protocol specifies the order of the messages involved in the conversation. |
| **data flow diagram** | A diagram that shows model elements (e.g., actors, logical components, hardware nodes) and the information that is exchanged between them. |
| **deployment diagram** | A diagram that shows the configuration of run-time processing nodes (hardware nodes) and the logical components, code components or processes that live on them. |
| **development view** | A view that describes the decomposition of the system’s code base into code components like files and libraries, dependencies between these code components, and the assignment of logical components of the logical view to code components. |
| **firmware architecture** | see *architecture*. |
| **hardware node** | A piece or several pieces of equipment that provide computational resources. |
| **interaction diagram** | A diagram that explains how several components collaborate for a specific purpose (e.g., a use case) by showing the components, their interactions, and the usual sequence of the interactions, optionally further explained or specified by comment or pseudo code. |
| **interaction** | An exchange of information or stimuli between two components. An interaction is normally modeled by a message sent from one component to another one. Interactions can be modeled on different abstraction levels and can hide more complex conversations. |
| **interface** | A set of services that specify all or part of the externally visible behavior of a component. One component can have several interfaces, and one interface can be implemented by different components. |
| **lexicon diagram** | A diagram showing the concepts and terms of a problem domain and the relationships between the concepts or terms. |
| **logical component** | see *component* |

| | |
|---|---|
| **logical view** | A view showing the decomposition of the system into logical components. |
| **mechanism** | A structure whereby components work together to provide some behavior that satisfies a requirement of the problem. |
| **message** | A specification of the conveyance of information to a component with the expectation that activity will ensue. Messages may be synchronous (procedure calls) or asynchronous (signals). |
| **object diagram** | A diagram that shows objects (e.g., logical component instances) and their relationships, often at a specific point in time. |
| **operation** | A service that can be requested from a component or system by an input interaction or message. An operation has a signature, which may restrict the actual parameters of the messages that are possible. |
| **passive object** | An object that has no thread of control of its own. |
| **physical view** | A view showing the architecture of the hardware nodes and how processes, logical components or code components map onto hardware nodes. It thus allows us to show how the software is deployed onto the hardware infrastructure. |
| **postcondition** | A truth-valued function which defines the required relation between the input and output values of an operation. |
| **precondition** | A truth-valued function which defines the possible inputs for which the operation guarantees the existence of a result. |
| **process view** | A view that describes the system's decomposition into executing tasks and processes by grouping threads and processes and by assigning logical structural elements to threads of control. |
| **protocol** | Defines all allowable sequences of services, interactions, or messages. |
| **scenario** | A purposeful set of interactions between some actors and the system or between the components of the system. |
| **sequence diagram** | An interaction diagram that shows the components and the messages exchanged between them. The interactions are arranged along the time axes. |
| **service protocol** | A specification of the constraints that apply to the order of the services offered by a component or its interface. |
| **service** | A behavior of a component for the benefit of and requested by a client (actor, collaborator) of that component. A service can be an operation of a use case. [23] |
| **signature** | The name and parameters (input and return parameters) of a service. |
| **software architecture** | see *architecture*. |
| **subsystem** | A component on a high abstraction level. Often a system is subdivided into several collaborating subsystems which again can be analyzed and modeled separately as systems (the collaborating subsystems become actors). |

[23] This definition of service is not to be confused with the usage of the term 'service' in the context of e-services and e-speak. There a service denotes a component, application or system being accessed over the e-speak environment. Like components and systems, an e-service has operations or use cases that are grouped together into one or several interfaces provided by the e-service.

| | |
|---|---|
| **system** | A composition of components organized to accomplish a set of specific functions. A system is thus the highest level component, the component of which the architecture is to be documented. |
| **use case diagram** | A diagram that shows the use cases and the actors of a system or component, and the relationships between actors and use cases. |
| **use case** | A service of a component or system specified by the interactions between one or several external actors and the component or system. A use case is often specified in several steps, each step triggered by an input interaction from an actor and containing some actions of the component. If on a given abstraction level the steps of a use case are not further decomposed then the steps are considered as being operations. In contrast to use cases, an operation only has one input interaction on the chosen abstraction level for interactions – the event or call that triggers the operation. Each input interaction triggers a new operation; any more detailed communication between actor and system is abstracted away. |
| **view** | A representation of a system from the perspective of a related set of interests or concerns. |
| **viewpoint** | A specification of the conventions for constructing and using a view. A viewpoint acts as a template from which to develop individual views by establishing the purpose and audience for a view and the techniques for its creation and analysis. |