ARCADE

An Open Architectural Description Framework

Erlend Stav, Ståle Walderhaug, and Ulrik Johansen

Developed by SINTEF ICT

SINTEF

Document information:

Title: "ARCADE: An Open Architectural Description Framework"
Version: 1.02, December 2013
Authors: Erlend Stav, Ståle Walderhaug, and Ulrik Johansen
Cover picture: Babak Farshchian

Content

1 Introduction

1.1 What is ARCADE?

ARCADE is a domain and technology independent architectural description framework for software intensive systems.

In this document we use the following definitions from [IEEE 1471-2000] of the central concepts of architecture and architectural description:

- **Architecture:** 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.
- **Architectural Description (AD)**: A collection of products to document an architecture.

The intended readers of this document are architects of software systems. We expect that the reader has a good understanding of object oriented and component based software development as well as the UML modelling language.

ARCADE can be used in different situations. When writing this document we have focused on support for the following uses:

- **Development**: Develop and document the architecture for a new system
- **Documentation**: Document the architecture of an existing system
- **Specialisation for domain**: Create a specialised architecture description framework for a domain (e.g. healthcare, Ambient Assisted Living)
- **Specialisation for application types**: Create an architectural description framework for a type of application (e.g. mobile systems, command and control information system)

For the two first uses in this list, a use guide is found in Chapter 5 of this document. To support the third and fourth use, Chapter 4 describes how to extend the framework. The generic framework described in Chapter 3 is the foundation for all the uses in this list.

1.2 Why use ARCADE?

ARCADE was created to assist in creating, understanding, and describing architecture of software systems. An architectural description of a software system makes it easier for software developers to:

- **Understand** the architecture of an existing system
- **Maintain architectural consistency** of a system during development and maintenance
- **Integrate** other systems with the described system

An architectural description framework assists the software architects in creating the architecture of a system, and in documenting this architecture. ARCADE was developed to help the architect with:

- **Documentation structure and content**: ARCADE provides a structure that ensures that documentation of different systems developed using the framework will have a uniform structure and content.
- **Handling quality related concerns**: ARCADE presents a list of quality related concerns that the architect should consider when creating the architecture, and tells how to document concerns of particular importance to the architecture.
- **Reusability and maintainability**: Components are the units of reuse and maintenance. Description of components is a central part of Architectural Descriptions created using ARCADE, and the Reference Architecture is based on components.

1.3 ARCADE Overview

This section gives a brief overview of ARCADE including how it relates to specialisations for specific domains.

ARCADE defines a set of concepts, and prescribes how to use these in the architectural description of a system.

A central part of ARCADE is its definition of a set of viewpoints to use as part of the architectural description. Each viewpoint defines how a specific view of the target system shall be described, including which UML diagrams to use. The *context viewpoint* defines how to describe the environment of the system, including system stakeholders and interfaces to other systems, while the *requirement viewpoint* deals with description of functional and quality requirements of the system. The *component viewpoint* defines how to describe the decomposition of the system into components, including their interfaces, interaction, and information. The logical distribution of components is the topic of the *distribution viewpoint*, while the *realisation viewpoint* deals with how to describe the realisation of the subsystems, including their deployment.

In addition to the viewpoints ARCADE defines how to describe *concerns* of special importance to the system, e.g. security and quality-of-service. These concerns will need special attention within all or most of the viewpoints. ARCADE also defines a *reference architecture* which further guides development of new architectures and can be used to compare different architectures to each other.

ARCADE as described in this handbook is a generic architectural description framework that can be used to describe any software system. Due to its generic nature it contains only a limited set of guidelines. As Figure 1-1 shows, the ARCADE handbook can be reused directly in when extending the ARCADE framework for a specific domain or for a specific application type. An important part of a domain specialisation is how domain standards and

other regulations influence the architecture and its documentation. The information contained in a domain specialisation should be a result of general requirements related to the specific domains and the experiences in the pilot development work. Healthcare ARCADE and Mobile ARCADE are examples of domain and application type specialisations of ARCADE.

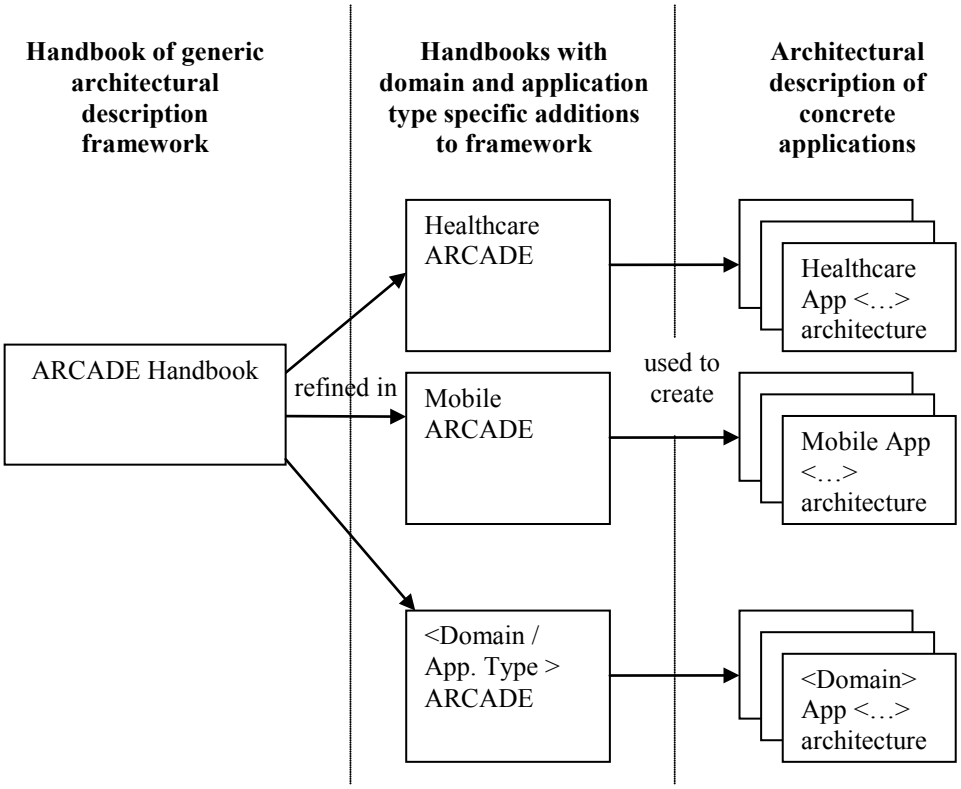

Figure 1-1 Overview of ARCADE, domain specialisations and use

1.4 Background

Model-based Architecture description Framework for Information Integration Abstraction (MAFIIA) is an architectural description framework for software intensive systems with a specialisation towards Information Integration Systems (IIS). The MAFIIA framework was developed in 2003 by SINTEF ICT, and has been used in several projects by SINTEF, but was not released in the public domain.

The ARCADE framework has used parts of the MAFIIA framework and handbook as a starting point for developing this handbook. The specialisation for Information Integration Systems (IIS) of MAFIIA has not been transferred to ARCADE, but may later be released as an application type specific specialisation for ARCADE.

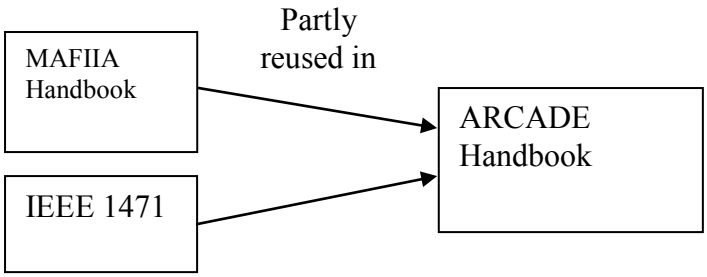

Figure 1-2 Background influences on ARCADE

ARCADE has also been influenced by IEEE 1471 ([IEEE 1471-2000]), which is a standard of recommended practice for describing the architecture of software-intensive systems. It defines a set of terms and a conceptual framework for architectural description. No specific notation is required by the standard, but it defines a set of elements that is required for an architectural description to conform to the standard (from IEEE 1471):

- *Architectural description identification, version, and overview information*
- *Identification of the system stakeholders and their concerns judged to be relevant to the architecture*
- *Specification of each viewpoint that has been selected to organize the representation of the architecture and the rationale for those selections*
- *One or more architectural views*
- *A record of all known inconsistencies among the architectural description's required constituents*
- *A rationale for selection of the architecture*

Compared to IEEE 1471, ARCADE gives further normative guidelines, including the use of UML as notation, and it defines a set of predefined (library) viewpoints and a reference model. The terminology and concepts defined in the standard are used in ARCADE, and architectural descriptions following the ARCADE framework should also conform to IEEE 1471.

1.5 Document Content

This document describes how to create the architectural description of a system. Chapter 2 describes the overall concepts and terminology for the architectural description, while the generic aspects of the description are handled in Chapter 3. Application Domain specific documentation, such as healthcare system reference architecture, is not included in this document. However, a description of what shall be included in such descriptions is documented in Chapter 4. In Chapter 5 guidelines for using ARCADE are presented, with main focus on describing the architecture of a new system. A brief section on how to describe existing systems is also included.

1.6 Terminology

1.6.1 Abbreviations

AD	Architectural Description
ARCADE	Model-based Architecture Framework for Information Integration Abstraction
UML	Unified Modelling Language

1.6.2 Terminology

The following table contains terms used in this document and their definition. For the underlined terms the definition is originally found in [IEEE 1471-2000].

Acquirer	An organization that procures a system, software product, or software services from a supplier. (The acquirer could be a buyer, customer, owner, user, or producer).
Application type	A category of applications grouped by the type of domain independent functionality they provide. E.g. information integration systems, command and control information systems, etc.
Architect	The person, team, or organization responsible for systems architecture.
Architecting	The activities of defining, documenting, maintaining, improving and certifying proper implementation of an architecture.
Architectural description	A collection of products to document an architecture.
Architecture	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.
Component	See: software component
Customer	The buyer of the Target System
Environment	The environment includes everything that is not a part of the target system, and which interfaces the target system directly. This includes both stakeholders and other systems.
Life cycle model	A framework containing the processes, activities, and tasks involved in the development, operation, and maintenance of a software product, which spans the life of the system from the definition of its requirements to the termination of its use.
Node	Computer interconnected to a data network

Reference architecture	A high-level, generic architecture which is used as the basis for development of concrete system architectures, and to compare architectures of existing systems to each other.
Software component	A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. (from [Szyperski 1998])
Subsystem	A coarse grained component that can also be regarded as a system.
System	A collection of components organized to accomplish a specific function or set of functions.
System stakeholder	An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system.
Target system	The system for which the architectural description is created
Vendor	The producer of the Target System
View	A representation of a whole system from the perspective of a related set of concerns
Viewpoint	A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purpose and audience for a view and the techniques for its creation and analysis.

2 Concepts

2.1 ARCADE Architectural Descriptions

Architectural descriptions are used to describe and document the environment for systems, the systems themselves, or part of systems. By "system" we here mean software intensive systems.

The description shall document all aspects of interest of a specific system, called the Target system, and its environment. There are other architecture levels such as business and subsystem architecture, but ARCADE will only cover the system's architecture. The architectural description is intended to cover those aspects that are necessary to make a complete documentation of a system's architecture.

The following set of concepts is applied in an architectural description:

- Viewpoints and views
- Stakeholders and roles
- Concerns
- System assets
- Reference architecture
- Modelling language

Each of the concepts is detailed in separate sections of this chapter.

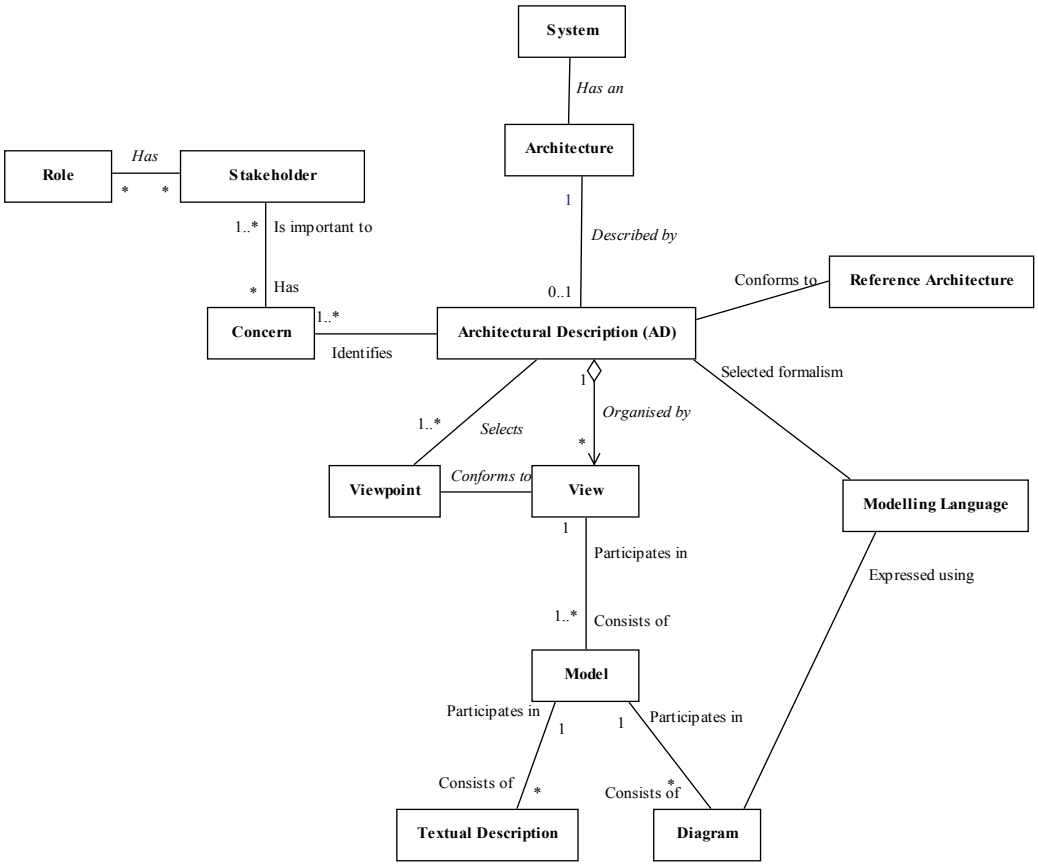

Figure 2-1 ARCADE concepts and their interrelations (based on [IEEE 1471])

Figure 2-1 identifies and shows what relations exist between the different ARCADE concepts. The figure shows that multiple concerns, viewpoints, views and models together constitute a complete architectural description. A model can consist of a number of textual descriptions and diagrams.

2.2 Viewpoints and Views

Viewpoints are used to create a view. The view consists of one or more models that describe and present different aspects related to structure and behaviour for a target system. An architectural description is a set of views created for a system. The difference between viewpoint and view can be described as:

- A viewpoint is a way of looking at a system,
- A view is what you see when looking from the chosen viewpoint.

This implies that ARCADE viewpoints are specified in this document, while views are created when documenting some target system.

Five different viewpoints are defined in ARCADE. They are:

- **Context Viewpoint**: The purpose with the context view is to describe all aspects of the Target System's environment, which is of importance to be able to document all

the interfaces between the Target System and its environment, and what the Target System is intended to do in its environment.

- **Requirements Viewpoint**: The purpose of the requirement view is to document all specific requirements related to the Target System.
- **Component Viewpoint**: The purpose of the component view is to identify and document specific physical or logical components. Component descriptions should be purely functional, described by their data, interfaces and functionality. Note that existing or predefined hardware- or software-units can be treated as components and included as components in the component view.
- **Distribution Viewpoint**: The purpose of the distribution view is to describe the logical distribution of software and hardware components. The distribution view shows if some components cannot be separated and if any must be separated.
- **Realisation Viewpoint**: The purpose of the realisation view is to describe any constraints on how the target system's components should be implemented and deployed into its environment.

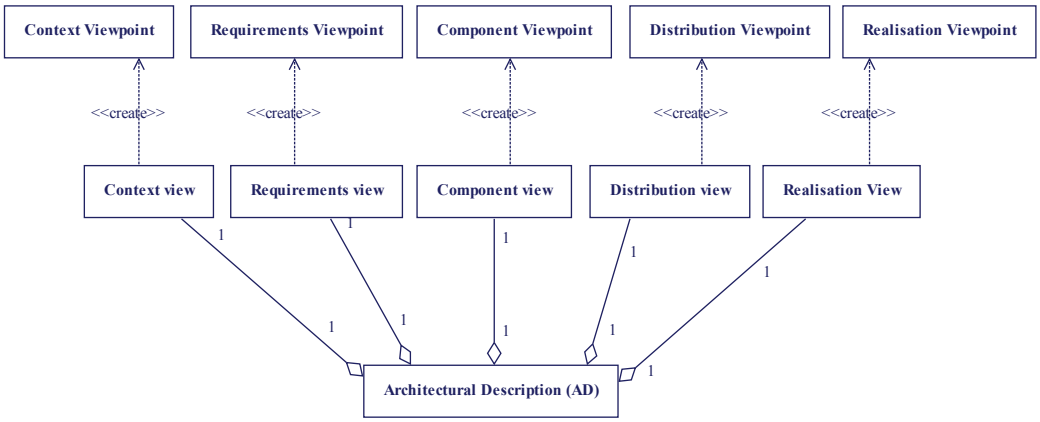

Figure 2-2 Different viewpoints and their interrelations

Figure 2-2 defines the different viewpoints and views, and shows relations between viewpoints, views, and the architectural description.

Each viewpoint may define one or more different expression forms, where the degree of formal or informal expression may vary. Some standardised modelling language should be used for formalised expression form (see section 2.7).

The five defined viewpoints are based on generic viewpoint properties, and this document defines how each of the five viewpoints shall be described, and the result of each of these descriptions shall be a corresponding view documentation. The totality of view descriptions contributes to the architectural description for the target system. The architect must take care to avoid inconsistencies and information redundancy between the different views of an architectural description.

Figure 2-3 shows the high-level process model for how the different views are developed.

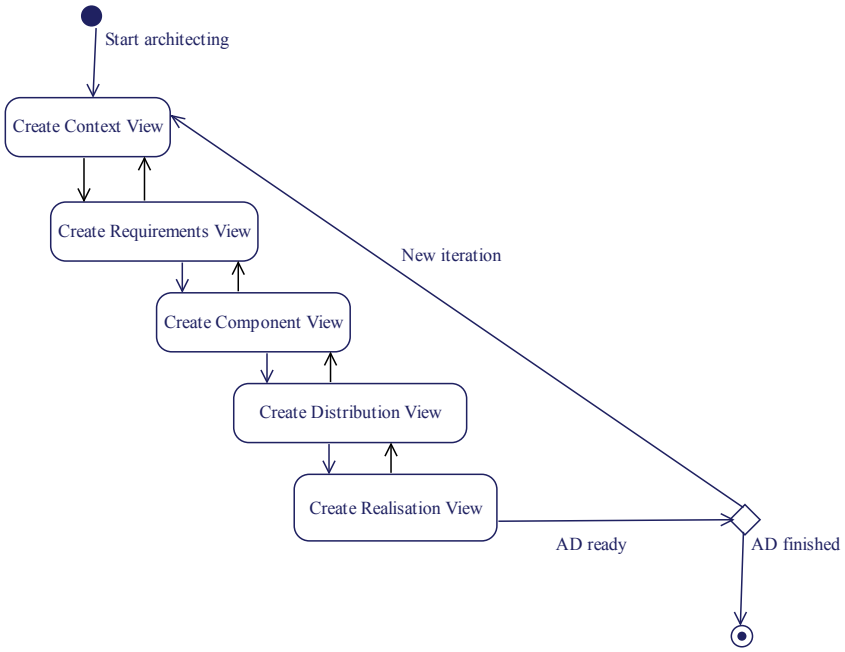

Figure 2-3 Process model for documentation of different views

The figure indicates that the context view (i.e. "highest level" view) is to be documented first, then requirement-, component-, distribution-, and realisation-view (i.e. "lowest level" view) at last. The view need not be finished before the next view can be documented, thus it is completely legal to partly document a view and then continue with the "lower level" views, and then later on continue or complete the partially done views. However, it is not recommended to first document in a "lower level" view something not treated in any "higher level" view. Further details on the iterative and incremental process suggested by ARCADE, is found in Chapter 6.

2.3 Stakeholders and their roles

A system has one or more stakeholders. Each stakeholder typically has interests in, or concerns related to, that system. Stakeholders have various roles with regard to the creation and use of architectural descriptions. Stakeholders include clients, users, the architect, developers and evaluators. As described in IEEE1471 [IEEE1471-2000], stakeholders typically possess one of two key roles, namely the acquirer (client) and the architect.

"The architect develops and maintains an architecture for a system to satisfy the acquirer. The architect may work from requirements provided by an acquirer or may be responsible for eliciting and developing requirements as a part of the architecture development. The role of the acquirer can be filled by a buyer, customer, owner, user, or purchaser". [IEEE1471-2000]

An acquirer can fill more than one role, e.g. a company can be both buyer and user.

2.4 Concerns

Concerns are related to the documentation of the functional aspects of the target system and its environment. Functional aspects that are considered to be of such importance that it should be treated separately and be specifically visible in the documentation should be identified and treated as a concern. A concern is visible and treated in relation to any view. Concerns are related to functionality and may be grouped into two main groups. These are:

- **Application specific functionality concerns**
  This group covers the functionality which will be necessary to implement if we consider an ideal world with no performance problems, network limitations, system failure and so on. This functionality will always be considered and described for a system, because else there will not be a system that implements what the users need.
- **Quality related functionality concerns**
  This group covers all types of functionality that may be used to improve the quality of a system according to what is required. These are concerns, which explicitly shall be visible as separate parts in some or all of the views. They will be included according to the complexity and desired quality for a system. For systems with high complexity and high quality requirements a wide range of such concerns will be explicitly defined and handled. This includes a set of different aspects that will contribute to some quality improvement of the target system.

2.5 System Assets

System assets are sources of information that can be used when developing the architecture descriptions. System assets can be considered as implicit requirements, which are not necessary to include in the requirement view, however assets may be included in component, deployment and realisation views. Examples of assets that are available are:

- **Dictionary**: A dictionary is a reference list of concepts important to a particular model aspect or concern along with discussion and/or definition of their meanings and applications.
- **Standards**: A standard is a formalised model or example developed by a standardisation organisation or established by general consent. When implementing e.g. a specific application type or domain specific type a set of various standards will probably be used, and should thus be referenced or documented.
- **Patterns**: A pattern is a description of a recurring, well-known problem and a suggested solution. Patterns are identified and can be used on many system levels.

2.6 Reference Architecture

A reference architecture is a high-level, generic architecture which is used as the basis for development of concrete system architectures, and to compare architectures of existing systems to each other. Reference architectures are structured into components and interfaces between these components. Figure 2-4 shows the top level and completely generic reference architecture.

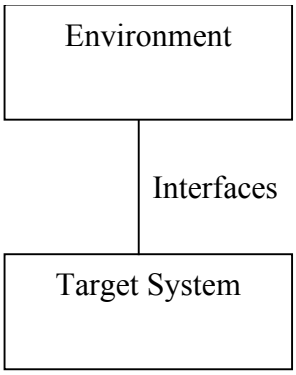

Figure 2-4 Generic reference architecture

This architecture differs only between what is the target system, what is the target system's environment and what are the interfaces between the target system and its environment. The main reason for this architecture is the documentation model used, where the environment is documented in the context view and the target system itself is documented in all other views.

2.7 Modelling Language

Documentation related to different views in the target system architecture can be more or less formally structured. Textual descriptions and sketches can be used where it is not possible, reasonable or efficient to use formal expression forms.

UML shall be used as formal description modelling language. All diagram types can in principle be used. However, guidelines about which diagram types to be used for each viewpoint model and how they shall be applied will be given in Chapter 3.

Other modelling languages could have been used but how they should have been applied is not considered in this document.

3 The ARCADE Framework

3.1 Introduction

This chapter presents the domain- and application type independent parts of the ARCADE architectural description framework. Specialisation of the description framework according to application type is done in Chapter 4, while specialisations according to specific domains will be done in separate documents. The specification of how this shall be done is provided in Chapter 5. The framework builds on the concepts that were defined in Chapter 2, and uses the terminology found in the last section of Chapter 1.

The next section of this chapter presents a list of concerns that can be of importance when defining an architecture. When creating an architectural description the architect should decide which of these concerns are of particular importance for the target system, and the section presents how the selected concerns shall be included in the AD. A section on model assets and how to describe these follows after this.

The generic version of the reference architecture for component based systems used in ARCADE is described next. The reference architecture is referred to in the description of each of the five ARCADE viewpoints: context, requirements, component, distribution, and realisation. These viewpoints are presented in the sections following the reference architecture. An architectural description based on ARCADE shall contain views created from each of the five viewpoints. Each viewpoint defines how to create a view consisting of a set of models. For each model, the purpose of the model and the notation to use is described. Some of the viewpoints descriptions also present specific architectural choices that should be documented in an AD, and for some of these choices a set of options are listed. Recommendations for architectural design related to the viewpoints are also included.

3.2 Concerns

Concerns are related to functionality and grouped into two main groups as defined in Chapter 2.

This section shall only identify relevant concerns, and describe those parts of each concern that is independent of Viewpoint. The parts of each concern that is specifically related to separate Viewpoints shall be described where relevant.

Architectural options and recommendations

Concerns relevant for the Generic System Architecture are identified and described in the following:

Application specific functionality concerns

This concern represents the application functionality for the target system and is implicit in the different model views, and is therefore not further grouped here.

Quality related functionality concerns

The following is a alphabetically sorted list of quality related concerns, which may be applied more or less for the specific target systems, depending on what the requirements are. Concerns, which shall be applied, shall be visible and treated specifically in the some or all of the views.

- **Communication**: The communication aspects shall be treated separately.
- **Concurrency**: This concern implies to have the possibilities to do multiple tasks in parallel. It is always relevant if a system has multiple simultaneous users, or machine or network controlled more or less autonomy tasks.
- **Configurability**: This concern is relevant for all systems having a certain complexity, and where adaptability and flexibility is important.
- **Distribution transparency**: This concern deals with how complexity related to distribution is handled. The concern includes the transparencies defined in RM-ODP [RM-ODP 1995].
- **Fault handling**: This concerns the system's ability to detect and serve the occurrence of unexpected or undesirable events in a controlled way. Fault handling normally includes both fault detection and fault handling.
- **Flexibility**: This concerns the system design, implementation, configurability and adaptability to its operational environment.
- **Interoperability**: The target system's ability to exchange information with other systems and to mutually use the exchanged information.
- **Maintainability**: This concerns the system design and implementation, and how easy it is to maintain these.
- **Naming**: Naming rules for objects and object attributes may be important, especially with respect to be able to obtain unique identification, but also to be used to obtain built in semantic rules used for addressing and localisation of objects.
- **Performance**: This concerns the operational system ability to perform the tasks it shall perform within quantified response times and serve times.
- **QoS**: The target system's ability to perform specified tasks according to a set of defined quantified quality parameters. Such parameters may relate to e.g. reliability and performance.
- **Reliability**: This concerns the target system's operability properties, i.e. can the system be trusted concerning the tasks it performs, its protection against potential threats and failure rate.
- **Replication**: This concern implies to keep multiple copies of information for some reason. May be done to improve performance or to assure availability of information.

- **Safety**: If the target system may represent a threat concerning persons' life and health then safety may be a relevant concern. Safety includes to do risk analysis to find out the probability that the target system may be responsible for the initiation of incidents, and also to determine the severity of such incidents.
- **Scalability**: The ease with which a system or component can be modified to fit the problem area.
- **Security**: The information served by the target system shall be protected. Such protection may e.g. imply control of users' access to information, classification of information according to security level and category. These problems can be solved through the implementation of confidentiality, integrity and accessibility.
- **Synchronisation**: Because the execution of a task normally takes some time and also involves the update of multiple atomic data element, it is necessary to synchronise different tasks operating on the same data.

3.3 System Assets

The concepts of System Assets and main categorisation are described in Chapter 2. System Assets are identified and grouped according to their *category* and a unique *name* (to make each asset referable), an additional *description* for each asset, and the specification whether the asset is mandatory or optional.

Assets are documented using textual descriptions and diagrams using e.g. UML [UML 1999].

Architectural options and recommendations

Assets identified for the generic architectural description framework are specified in the following:

Dictionary

There a various kinds of dictionaries that can help to define a common terminology and understanding of concepts:

- **Thesauri**: A thesaurus is a list of terms used for a certain application or domain. A thesaurus is intended to be complete for its domain and can also contain a list of synonyms for each preferred term. A thesaurus is also called a "controlled vocabulary".
- **Nomenclatures**: A nomenclature assigns codes to domain concepts. Concepts can be combined according to specific rules to form more complex concepts. This leads to a large number of possible code combinations.
- **Ontologies**: A common way to express a dictionary in a domain is to use an ontology. [Gruber 1993] has defined ontology as: "*An ontology is a formal explicit specification of a shared conceptualisation*". A conceptualisation, in this context, refers to an abstract model of how people think about things within the target domain. In large domains. There may be many ontologies covering parts of the domain.

The list of terms in section 1.6.2 is a Thesauri for the concepts of the generic ARCADE description framework.

Standards

- **UML 1.4 (mandatory)**
  The language to be used as "formal" expression form for the different views in the architecture description documentation.

Patterns

Due to the generic nature of the description framework almost any proven pattern can be of interest to some target system. The set of documented patterns is too broad to present here, and instead we refer architect to sources such as [Gamma 1995], [Buschmann 1996], and [Schmidt 2000].

3.4 Reference Architecture

The reference architecture shall be shown as a set of components with the relations between these components. Any component in a reference architecture can be further decomposed one or more levels. A textual description of each component and interconnection shall be given.

Architectural options and recommendations

Figure 3-1 shows the generic reference architecture of ARCADE. This reference architecture can be specialised and refined for specific application types and specific domains. If the architecture to be used in a target system is not in correspondence with Figure 3-1, the reasons for the mismatch should be documented.

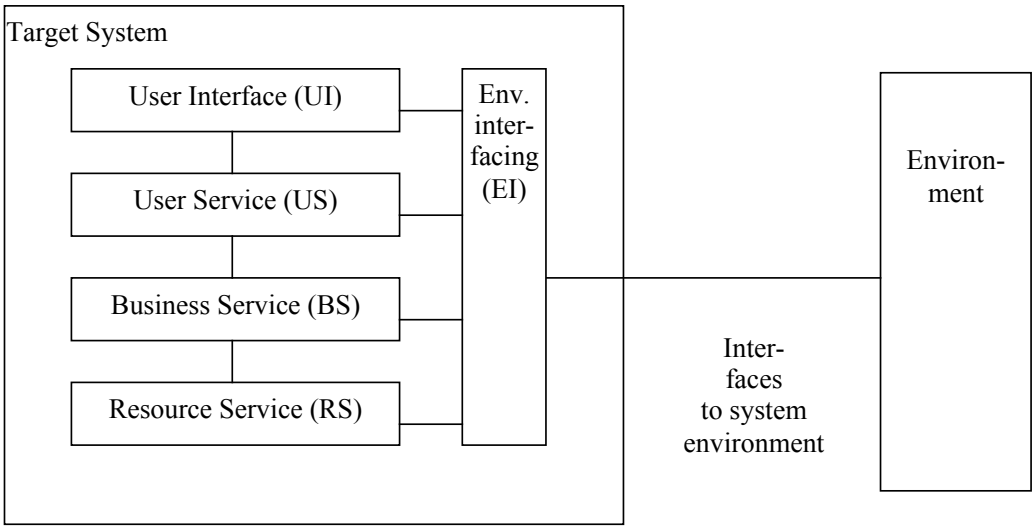

Figure 3-1 Generic Reference System Architecture

As the figure shows, the reference architecture divides the target system into a set of logical tiers, and defines how the system interfaces with the environment. A tier is a logical partitioning of a system where each tier has a unique responsibility. Each tier is loosely coupled to the tiers directly above and below, as illustrated by the connections in the figure. In a specific target system one or more of the tiers in the figure may be irrelevant and can be omitted, e.g. systems without a user interface have no user interface tier and may have no user service tier.

The tiers defined in the reference architecture are (inspired by [COMBINE 2002]):

- **User Interface (UI) tier**: The user interface tier provides user interaction logic through a set of user interface components. It communicates with the user service tier.
- **User Service (US) tier**: The user service tier provides the user's model, which may include user session logic and user-side representations of processes and information. It is an abstraction for a set of business services, making the business service provision transparent for the user interface tier. The components at this tier can be divided broadly into **entity** and **service** components, where the entities represent information as it is presented to users.
- **Business Service (BS) tier**: The business service tier provides service components that represent business functionality and also pervasive functionality. This tier provides enterprise-level services, and is also responsible for protecting the integrity of enterprise resources at the business logic level. The components at this tier can be divided in **entity** and **service** components, where the entities directly represent persistent business information.
- **Resource Service (RS) tier**: The resource service tier provides global persistence services, typically in the form of databases. Resource adapters (e.g. JDBC or ODBC drivers) provide access, search and update services to databases and its data stored in a database management system (DBMS) like Oracle or Sybase.

The **Environment Interfacing** (EI) handles the interfacing to and from the environment. As the figure shows, the system can interface with the environment from any tier. Usually the business service tier or the resource service tier is the preferred tier for interfacing with other systems in the environment, but e.g. legacy systems may limit what is technically and economically achievable. Performance, security, and other quality related functionality concerns will also influence this interfacing. Normally the EI is not used for communication between the tiers of the target system, as this kind of communication usually uses the direct connections between adjacent tiers.

Each tier in the system can use the components defined in the tier that is directly below it through their interfaces. This interaction can take many forms, e.g. synchronous or asynchronous calls or messages, and will use the invocation mechanisms provided by the selected component infrastructure. Control may in some cases also need to flow from a component to the tier directly above. To avoid direct dependencies on the tier above it is preferable in these cases to use some kind of notification mechanism (typically using the event model of the selected component infrastructure), but other kinds of interaction may in some cases be required.

3.5 Context Viewpoint

Purpose	Describe the environment to the target system in terms of its business-related aspects, other involved technical systems and the mapping of business aspects to the target system.
Stakeholders addressed	Acquirer (buyer, customer, owner, user or purchaser) and architects
Required models	Business aspects model, Environment system model, Business to system mapping model

Optional models	None

The context view shall only document the environment of the target system. The environment includes everything that is not a part of the target system, and which interfaces the target system directly. This includes both stakeholders and other systems.

No documentation specific for the target system itself shall be described here. Note that the context view work shall not develop detailed documentation about the systems and processes that constitutes the environment. It shall only collect and organise such documentation according to how context view documentation shall be done. If detailed information about the environment is missing (e.g. detailed documentation of interfaces to technical systems to be interfaced to the target system) that documentation must be developed. However, the developed documentation is not a part of the target system's architectural description.

Several of the defined diagram types, *Textual descriptions*, *Structure*-, *Behaviour*-, *Interaction*-, and *Implementation*-diagrams (see Chapter 7) are recommended used in the documentation of the different models in the context view. Textual descriptions should only be used where the more formally diagram types do not fit in.

The context view documentation is composed of three different models as specified in the following sections.

3.5.1 Business Aspects Model

Purpose	Shall document any business related concern that will increase the understanding of what problems the target system shall solve, or what functionality it shall implement.
Input	Customer supplied information (i.e. requirement to target system, business related information)
Output	Textual description, UML class diagram, UML use case diagram, UML collaboration diagram

The Business aspects model shall only document aspects related to the parties involved in operating and using the target system and the processes to be carried out by the involved parties. The Business aspect model shall not document specific existing or planned technical solutions, neither shall it document mapping of business related aspects to any technical solution.

The Business aspects model includes the documentation of *information*, *stakeholders*, *processes*, and *relations* between these, independent of what technical solutions are involved in the realisation of this business model (UML class diagrams, UML use case diagram, UML collaboration diagram).

3.5.2 Environment Systems Model

Purpose	Shall document other technical systems (environment systems) that will be involved in the implementation of the Business model, or influences the operation of the target system.
Input	Customer supplied information (i.e. target system requirement specifications, environment system documentation)
Output	Textual description, UML sequence diagram, UML use case diagram, UML collaboration diagram

The Environment systems model shall identify and organise existing documentation of environment systems that will be involved in, or influences the operation of the target system. Only documentation of importance to the target system is relevant.

Documentation of *Interfaces* for each system should be identified (e.g. using high level UML sequence diagrams) in addition to *collaborations* between environment systems, and between environment systems and the target system (UML use case diagram and/or UML Collaboration diagrams).

3.5.3 Business to System Mapping Model

Purpose	Shall document what parts of the documented Business aspects model are mapped to technical solutions constituted by environment systems and the target system, and how the different parts of the business aspect model is mapped to the different involved systems.
Input	Customer supplied information (i.e. target system requirement specifications, requirements/needs related to the business aspects)
Output	Textual description

There is a need to specify how the technical independent Business aspect model shall be mapped to technical solutions. Some parts of the business modes shall probably still be implemented by some work performed by persons, while some parts of the business model shall be implemented by technical solutions, together constituted by identified environment systems and the target system.

Business to system mapping model will include the documentation of what *information* and *processes* are mapped to which systems, and which *stakeholders* are involved in operating and using the different involved systems.

3.6 Requirement Viewpoint

Purpose	Identify documentation of, or specify requirements related to any concerns to the target system. Requirements shall be testable and shall be used to verify that the target system is able to perform its intended tasks.
Stakeholders addressed	Acquirer (buyer, customer, owner, user or purchaser) and architects
Required models	Requirement model, Target system interface model
Optional models	

The requirement view shall document uniquely identifiable and testable requirements. The main basis for the requirements is the context view documentation, which implicit represents requirements to the target system. Each requirement is documented as a separate entity and may concern any aspect of the architecture description.

The requirement view documentation consists of two models as specified in the following sections.

3.6.1 Requirement Model

Purpose	Shall be complete in identifying, and eventually specifying all relevant requirements to the target system where a requirement shall be verified.
Input	Context view
Output	Textual description (Requirement specification form), UML sequence diagram, UML use case diagram, UML collaboration diagram

The requirements may be related to any aspect of the target system and its environment. Requirements may be related to any viewpoint, concern, assets, architecture description, detailing level, implementation aspects, and other. Requirements shall be uniquely identifiable and they shall be testable. This assures that the target system can be verified against the requirements set to it.

The requirements can be hierarchically decomposed one or more levels. Normally only the leaf level specifies real requirements, while node levels only represent a collection of requirements at lower level. The top level structuring of requirements shall be according to ARCADE concept descriptions in Chapter 2.

The requirements specified in the ARCADE are written with flexibility in mind. These requirements define a set of requirement alternatives rather than specific requirements that can be selected among when writing the requirement view for an architecture description for a specific target system.

The requirements view is documented using textual descriptions and UML diagrams (preferably use case diagrams). A specific textual description form (i.e. the Requirement specification form in Table 3-1) shall be used, and shall include all defined requirements. However, the detailing and specific documentation of the requirement may be done using other diagram types. The textual form shall at least for each requirement specify a unique identifier for the requirement, a specification of the requirement, which is sufficient to be able to verify it, and references to supplementary requirement descriptions if any. A priority attribute is recommended as some requirements may be of less importance than other. Where hierarchical decomposition of requirements is used, that shall also be expressed. It is recommended to use the requirement identifier as a mean to express this.

Table 3-1 Requirement specification form

Req. id	Req. description	Acceptance test	Priority {high | medium | low}
R1	…	…	…
Ri.j.k	…	…	…

A specific short form notation is used to express the requirements: Values “a”, “b” and “c” defines parts of a requirement. The term {a|b|c} specifies that either of the alternatives a,b or c is a valid requirement; {a+b+c} specifies that the set with any combination of a, b and c is a valid requirement; and {a*b*c} specifies that the set where all alternatives a, b and c must be included to define a valid requirement.

Architectural options and recommendations

Table 3-2 defines a set of requirement choices, valid for a Generic system architecture, to be applied as a guideline for defining specific requirements in ADs. The table represents a template that defines the top level structuring of the requirements, but where some requirements are not specified or relevant for the Generic system architecture (denoted “none”). An extended and completed version of the same table should be used for application type specific requirement and for domain specific requirements.

Table 3-2 Requirement choices for the Generic system architecture

Req. id	Requirement identifications
R1	**Viewpoint related requirements**
R1.1	*Context view (none)*
R1.2	*Requirements view (none)*
R1.3	*Components view (none)*
R1.4	*Distribution view (none)*
R1.5	*Realisation view (none)*

Req. id	Requirement identifications
R2	**Concern related requirements**
R2.1	*Application functionality concerns (none)*
R2.2	*Quality related concerns*
R2.2.1	*Requirement*: Support quality related concerns {Communication + Concurrency + Configurability + Distribution Transparency + Fault Handling + Flexibility + Interoperability + Maintainability + Naming + Performance + QoS + Reliability + Replication + Safety + Security + Scalability +Synchronisation....}
R2.2.1.1	*Requirement*: "Communication" obtained through...
R2.2.1.2	*Requirement*: "Concurrency" obtained through...
R2.2.1.3	*Requirement*: "Configurability" obtained through {Data Based Configuration +...}
R2.2.1.4	*Requirement*: "Distribution Transparency" obtained through...
R2.2.1.5	*Requirement*: "Fault Handling" has functionality {Serve Report Request + Report Request Actions + Fault Log + Fault Reports + Fault Statistics}, and shall be {Always Available + Uniform Accessible} from anywhere within the System.
R2.2.1.6	*Requirement*: "Flexibility" obtained through {Modular Design + Data Based Configuration + Component Autonomity + Generalised Solutions +...}
R2.2.1.7	*Requirement*: "Interoperability" obtained through {Standardised Interfaces + Common Uniform Interface}
R2.2.1.8	*Requirement*: "Maintainability" obtained through...
R2.2.1.9	*Requirement*: "Naming" obtained through...
R2.2.1.10	*Requirement*: "Performance" obtained through {No Data Based Configuration + Non Distribution + Optimised Solutions}
R2.2.1.11	*Requirement*: "QoS" obtained through...
R2.2.1.12	*Requirement*: "Reliability" obtained through {Use Of DBMS + Duplication + Distributed Solution +...}
R2.2.1.13	*Requirement*: "Replication" obtained through...
R2.2.1.14	*Requirement*: "Safety" obtained through...
R2.2.1.15	*Requirement*: "Security" obtained through...
R2.2.1.16	*Requirement*: "Scalability" obtained through {Distributed Solution + ...}
R2.2.1.16	*Requirement*: "Synchronisation" obtained through...
R3	**Model assets related requirements**

Req. id	Requirement identifications
R3.1	*Dictionary*
R3.2	*Standards*
R3.2.1	*Requirement: Use some standardised modelling language for formal type of documentation of Ads*
R3.3	*Patterns (none)*
R4	**Reference architecture related requirements** (see Figure 3-1)
R4.1	*Environment*
R4.1.1	Environment systems (none)
R4.1.2	Environment systems interface (none)
R4.2	*Target System*
R4.2.1	Target System configurations
R4.2.1.1	*Requirement*: UI shall exist and be supplied from {US+EI}
R4.2.1.2	*Requirement*: US shall exist and be supplied from {BS+EI}
R4.2.1.3	*Requirement*: BS shall exist and be supplied from {RS+EI}
R4.2.1.4	*Requirement*: RS shall exist and be supplied from {BS+EI}
R4.2.1.5	*Requirement*: EI shall exist and be supplied from {US+BS+RS+EI}
R4.2.2	Target system interfaces (none)
R4.2.3	Functionality and data in target system components
R4.2.3.1	*Requirements*: UI functionality includes: - Implements GUI (i.e. interface between the System and the user) - Communication with {EI+US} - Mapping of information between {EI+US} and GUI *Comments*: -
R4.2.3.2	*Requirements*: US functionality includes: - Communication with {UI*{BS+EI}} - Mapping of information between UI and {BS+EI} *Comments*:

Req. id	Requirement identifications
	-
R4.2.3.3	*Requirements*: BS functionality includes: - Communication with {US*{RS+EI}} *Comments*: -
R4.2.3.4	*Requirements*: RS functionality includes: - Communication with {BS*EI} - Read-only, or read-and-write. Support of transactions if write - Replicated data *Comments*: -
R4.2.3.5	*Requirements*: EI functionality includes: - Communication with {Environment*{RS+BS+US+UI}} - Support of protocols {<list of>...} and communication mechanisms {<list of>...} - Transformation to common physical formats used in RS *Comments*: - List of protocols (i.e. <list of>) is specified in R4.1.2.3

3.6.2 Target System Interface Model

Purpose	To be a supplementary specification to the Requirement model in order to obtain a more complete and easier understandable specification of the target system's interfacing to its environment.
Input	Context view (Business to system mapping model), Requirement model
Output	UML sequence diagram, UML use case diagram, UML collaboration diagram

This supplementary requirement documentation shall only concern the target's system interfacing to its environment. The documentation must relate and conform to the business to system mapping model of the Context view and also one or more corresponding requirements in the Requirement model.

Target system interface model will include the documentation of *who* operated the interfaces, *actions* initiated and *responses* given, and the type of *functionality* performed as a result of executed operations (UML sequence diagram, UML use case diagram, UML collaboration diagram).

3.7 Component Viewpoint

Purpose	Describe the system in terms of its subsystems and information objects, and document how subsystem interaction and information processing is carried out in order to provide the desired behavioural effect
Stakeholders addressed	Architects
Required models	System information, System decomposition, System collaboration, Component and interface specification
Optional models	

The purpose of the component view is to describe the system in terms of its subsystems and information objects, and to document how subsystem interaction and information processing is carried out in order to provide the desired behavioural effect. Structuring a system into components may be done in many different ways. The argumentation for what a specific component shall include should be based on specific criteria. Such criteria may be that the component is already an existing physical software or hardware component, the functionality contained in the component represents a natural collection of functionality, or the interfaces between the component and its environment is standardised or well defined interfaces.

As part of a component view, it is recommended that models are created with special focus on information, system decomposition, system collaboration, and component and interface specification. In addition models for specific concerns can also be included.

It is preferable to keep the description of the component view at a functional level, and not deal with technology choices. Mapping to chosen technology is instead done in the realisation view. There may be cases, though, where this ideal is not fully achievable. The technologies of choice can have their own preferred approaches and architectural recommendations. If the component view is developed totally independent on the technology, the mapping to the realisation may be difficult and lead to solutions that are suboptimal. The technology independence of the component view is thus a guiding principle that can be deviated when good reasons exist. It is especially important to describe the rationale of the decisions in these cases.

3.7.1 System Information Model

Purpose	Specify the relationships between and properties of the central information objects in the system that must always be true (invariants)
Input	Requirements view
Output	UML class diagram, Textual description

The system information model specifies relationships between and properties of information objects that must always be true (invariants). The system information model shall be specified as a traditional information model using UML class diagrams. The system information model defines the information semantics by relating information objects and defining concepts that must be understood in a common way in the system (often referred to as "a shared data model").

Each class of information objects shall in addition be described with a textual description of its purpose, and a description of each of its public attributes that are of importance at the architectural level.

Related to the Reference Architecture the system information model primarily describes the entities of the business service tier and the user service tier.

3.7.2 System Decomposition Model

Purpose	Describe how the system is divided into different subsystems or components, and how these are related to form a coherent whole.
Input	Requirements view
Output	UML class diagram

The system decomposition model describes how the target system is divided into different subsystems or components, and how these are related to form a coherent whole. This includes how subsystems or components in the target system relate to subsystems or components in other systems. A modularised description of the system facilitates reuse of well-defined components and identifies points of integration with other systems.

Decomposition of the system will usually be applied hierarchically, with the initial diagram showing an overview of the whole system. Further decomposition of subsystems is performed when this is seen as part of the architecture, while decomposition of more fine grained components are typically considered as detailed design and not handled in the AD. In cases where a subsystem is particularly large, complex, or independent on the rest of the system, an alternative is to create a separate, complete AD for the subsystem.

As part of the decomposition model, the components are also related to their tier in the reference architecture. Decomposition models can be created for all of the tiers in the reference model, but usually at least the business service and user service tiers are described in this model.

3.7.3 System Collaboration Model

Purpose	Describe the main interactions in the system as a set of collaborating components

Input	Target system interface model (Requirements view)
Output	UML class diagram, UML activity diagram, UML sequence diagram, UML collaboration diagram, Textual description (area of concern)

The target system interface model of the requirements view described the functionality provided by the system to the users and other systems. The collaboration model includes description of how each of the identified actions is handled in the architecture. In addition to the collaborations originating in system interface requirements, the collaboration model also describes other mechanisms that are considered part of the architecture. For each collaboration to be described, the area of concern shall be given in a textual description. Class diagrams can be used to display the set of interacting components, along with the connections and interfaces directly involved in the interaction. The collaboration is further detailed by sequence, collaboration or activity diagrams. An alternative to focusing directly on components in the description of collaborations is to first identify and describe the roles involved, and later map these roles to the components that play these roles in the system.

3.7.4 Component and Interface Specification Model

Purpose	Describe each of the identified components and interfaces of the target system, including operation signatures and behaviour
Input	Requirements view. System decomposition model, System collaboration model (Component view).
Output	UML class diagram, UML state chart diagram, Textual description

This model describes the details of each interface and component that was identified in the decomposition and collaboration models.

All interfaces provided by one or more component of the system shall be documented. The description shall contain the purpose of the interface, and each operation in the interface shall be described with signature and a textual description. The interface description can also include rules on ordering of calls, and other pre- and post-conditions of operations. The behaviour expected of components implementing the interface can be further specified by state chart diagrams.

For each component in the system there shall be a class diagram or textual description of the interfaces the component provides. If the component defines further attributes and operations, a description of these shall be included. All dependencies of a component on other components and interfaces shall also be described, either in a textual description or in a class diagram. Pre- and post-conditions, and state chart diagrams can also be used to specify behaviour.

3.8 Distribution Viewpoint

Purpose	Shall describe the logical distribution of system components and document which components that must be separated and which that cannot.
Stakeholders addressed	Architects
Required models	System Distribution Model
Optional models	Role Distribution Model

The distribution viewpoint provides conventions on how to create a distribution view for the target system. The distribution view has models that describe how the system components are logically distributed, that is, independent of existing infrastructure, which is considered in the realisation viewpoint.

To document the view, two description models are suggested, where one is optional. To decide if a model is required, one need to analyse the target system's properties and the concerns specified. If the target system has low complexity with limited functionality, complexity and the environment components are centralised, the optional model can be skipped. However, if the target system has more than one owner or involves more than one role, the "Role distribution model" must be included.

It is recommended to model the logical distribution using UML deployment diagrams. Deployment diagrams describe the configuration of processing resource elements (hardware) and the mapping of software implementation components onto them. These diagrams contain components and nodes, which represent processing or computational resources, including computers, printers and so forth.

To ensure that the distribution view is in accordance with the target system's mission, all preceding views in the process must be taken into account. Related to the Reference Architecture, components at each tier can be distributed, though it is recommended not to separate tightly connected components in a tier. The defined system concerns will be used as a precept through the architecture process.

3.8.1 System Distribution Model

Purpose	Shall describe logical units or components that must be distributed and deployed together.
Input	System Decomposition Model, System Collaboration Model (Component View)
Output	UML Deployment Diagram and textual descriptions (rationale)

In this model, the logical distribution of all components that are part of the target system must be described. To create the distribution model, one will need to consider the following environmental parameters:

- **System size**. The number of components and interfaces involved. A large system would normally require higher degree of distribution than a small system,
- **Geographical distribution of components**. The geographical location of interfacing environment components. If the target system components will interface existing components, this must be considered when creating the system distribution model.
- **Communication properties**. If communication lines between nodes in the system are slow, one should model distribution in order to optimise the target system performance. Components that require high bandwidth and low latency should not be distributed apart from each other.
- **Data processing capacity**. The components' need for data processing power varies a lot. Target system performance will depend on the distribution of processing intensive components. One should seek to distribute such components onto separate logical nodes to facilitate system load sharing.
- **Reference Architecture tier**. Components at each tier can be distributed independent of each other. However, one should seek to avoid separation of components in the user interface and user service tiers. In the same manner, components in the business tier should not be separated from resource tier components that form the basis for their operations.

In addition to these properties, requirements specified in the *Requirement View* may be directly or indirectly related to the distribution of components. If high performance is a requirement or a system concern, the system components should be distributed in a way so that this requirement can be met. Each category of requirements defined in the *Requirement View* can affect the distribution model.

The *Component View* describes the component decomposition and interaction. This information is of highest importance for the distribution models. As for general system engineering, we seek to minimise external coupling of components.

- **The system decomposition model** describes how the system is divided into different components or packages of components, and how these are related to form a coherent whole. Thus, it identifies the lowest level of distributable components, that is, which components or packages of components that actually can be distributed.
- **The system collaboration model** provides information about which components are interacting with which. Components that interact intensively with each other should not be distributed separately.

The target system's concerns influence the distribution model. An important concern with respect to distribution is transparencies. It is highly recommended to conform to the distribution transparencies defined in RM-ODP [RM-ODP 1995]. Using these transparencies, one will hide complexity related to component distribution from all tiers in the Reference Architecture. The transparencies defined in RM-ODP are:

- **Access transparency**, which masks differences in *data* representation and *invocation* mechanisms to enable interworking between objects. This distribution transparency solves many of the problems of interworking between heterogeneous systems, and will generally be provided by default.
- **Failure transparency**, which masks from an object the failure and possible *recovery* of other objects (or itself) to enable fault tolerance. When this distribution transparency is provided, the designer can work in an idealized world in which the corresponding class of failures does not occur.
- **Location transparency**, which masks the use of information about *location in space* when identifying and binding to interfaces. This distribution transparency provides a logical view of naming, independent of actual physical location.
- **Migration transparency**, which masks from an object the ability of a system to change the location of that object. Migration is often used to achieve load balancing and reduce latency.
- **Relocation transparency**, which masks relocation of an interface from other interfaces bound to it. Relocation allows system operation to continue even when migration or replacement of some objects creates temporary inconsistencies in the view seen by their users.
- **Replication transparency**, which masks the use of a group of mutually behaviorally compatible objects to support an interface. Replication is often used to enhance performance and availability.
- **Persistence transparency**, which masks from an object the *deactivation* and *reactivation* of other objects (or itself). Deactivation and reactivation are often used to maintain the *persistence* of an object when the system is unable to provide it with processing, storage and communication functions continuously.
- **Transaction transparency**, which masks coordination of activities amongst a configuration of objects to achieve consistency.

One can implement one or more of these transparencies, dependent of system requirements.

3.8.2 Role Distribution Model

Purpose	Describe the distribution of the different roles that are part of the target system.
Input	Business Aspects Model (Context View), System Distribution Model (Distribution View)
Output	UML Deployment diagram and textual descriptions (rationale)

For large systems with many stakeholders and roles involved, it is necessary to document the distribution of these roles and how they are related to each other and the target system. A role can be a person or a system component (or package of components). Components that are associated with a role should be distributed along with this role.

For example, if John Smith has the role as a System Administrator and is the only one that can operate component 1, component 1 and role System Administrator should be kept logically together.

3.9 Realisation Viewpoint

Purpose	Shall describe the realisation of the target system in terms of its subsystems. The view will describe how to structure implementation and deployment the target system.
Stakeholders addressed	Architects
Required models	System Deployment Model, System Integration Test Model,
Optional models	

The realisation view will document constraints on how the target system should be implemented and deployed into its environment. The deployment models will get input from the distribution view. There is not necessarily a direct mapping from the logical distribution created in the distribution view to the actual distribution that will be created in the realisation view. Two logically distributed components can be deployed to the same or different nodes. However, two components that are not logically distributed should always be deployed to the same node.

In this view, technological aspects such as component technology, collaboration mechanisms and system platforms will be taken into account.

Realisation of the target system includes implementation and deployment of components.

Implementation must consider:

- **Platform**: Relevant hardware and software platforms must be investigated and specified. There might be dependencies between hardware and software platforms that must be taken into account.
- **Component technology:** Related to platform, the architects must investigate and specify which component technology is available, required or most feasible. Mechanisms for component interaction, such as messaging or remote procedure calls, must also be considered.
- **Development process:** The architects should recommend the most suited development process. This process must be adapted to the target system's properties and the context in which it should be deployed.

Deployment of components must consider:

- **"All in once":** One needs to develop a strategy for deployment of target system components. This can be done step by step or all in once. In a distributed system, one must decide if deployment at different locations should be done in parallel or serially.

- **Fault handling:** One should develop a strategy for fault handling. Roles and responsibilities should be specified. This is to ensure that the realisation process can be carried out properly within specified time and cost limits.

To describe the realisation view, UML deployment diagrams and textual descriptions should be used. It is recommended to model the System Deployment Model first since the System Integration Test Model uses it as input.

3.9.1 System Deployment Model

Purpose	The purpose of this model is to describe the set of system deployment configurations.
Input	System Distribution Model (Distribution View), Requirement Model (Requirement View)
Output	UML Deployment model, textual descriptions

The deployment configurations describe the actual distribution showing the physical relationships among software and hardware components in the target system. The UML deployment diagrams will describe the physical nodes' connectivity for the system, where each deployment diagrams may highlight some parts of the target system.

3.9.2 Technology Mapping Model

Purpose	Shall describe a how system components maps to technological solutions, concepts and mechanisms
Input	Requirement model (Requirement View), Component and interface specification model (Component View), System Distribution Model (Distribution View), System Deployment Model (Realisation View)
Output	UML Component Diagrams, UML Deployment Diagrams, Textual descriptions (rationale)

This model describes how abstract functional components can be mapped into a concrete implementation in hardware and software. The problem is to bridge the gap between the abstract system and the concrete components in the real world. The models should specify how components are related to or implemented by technological solutions, concepts and mechanisms that are available or that should be developed.

3.9.3 System Integration Test Model

Purpose	Shall describe a set of test scenarios to be conducted during system deployment (subsystem integration).
Input	Requirement model (Requirement View), System Deployment Model (Realisation View)
Output	Textual descriptions

The test model describes how one can verify correct operation of the target system with a set of specified tests. The testing can be conducted at different level and on different parts of the target system. However, all test procedures and results must be documented in a test model.

The test scenarios can be described using informal textual descriptions or a standard formal notation for system testing.

4 Extending ARCADE

ARCADE can be applied to different application domains and application types. However, using ARCADE with a specific application domain or application type requires the adaptation of ARCADE for that purpose. Specialisations and refinements for a specific domain shall be documented in a separate document (<domain / application type> ARCADE), and this chapter shall specify the structure of, and what shall be contained in that document.

Application domains and application types may in principle be any domain or application type. At the moment, the following domains and application types have been identified: Mobile, Healthcare, Defence, Ambient Assisted Living (AAL), and Information Integration Systems (ISS). Separate documents (e.g. Mobile Arcade) will describe framework additions for the domain specific or application type specific aspects.

The issues to be described in these documents shall correspond to what has been done for the generic part of ARCADE in Chapter 3. A domain specific or application type specific description shall have the following structure:

1 Introduction
2 Concerns
3 System Assets
3.1 Dictionaries
3.2 Standards
3.3 Patterns
4 Viewpoints

The importance and size of each chapter and section of the domain specific document will depend on domain specific properties for the domain described. In most cases we think that the focus will be on identification and use of specific Concerns and System assets, while description concerning one or more of the defined Viewpoints may be of interest in certain situations.

Persons responsible for writing any "<domain / application type> ARCADE" document (i.e. ARCADE domain reference architecture specific designers) should use Chapter 3 and this chapter to get information about the form and type of content of the domain specific or application type specific document. Chapter 3 should be used to get information about generic technical related aspects which already have been covered, and need not be covered as domain or application type specific.

Those persons responsible for architecture documentation according to ARCADE (i.e. architecture designers/"documenters") should use Chapter 3 of this document and those "<domain / application type> ARCADE" specific documents relevant for the application to be documented.

5 ARCADE Use Guidelines

ARCADE may be used for different purposes, e.g. it may be used to document existing systems, it may be used for the documentation of an organisation's activities, and it may be used as a tool in the development of technical systems. This chapter gives guidelines for the use of ARCADE in development of new systems. It also contains some comments on how documentation of existing systems will deviate from these guidelines.

This document describes the generic ARCADE architectural description framework and its specialisation for information integration system. In addition, the architect should also select and use, or if necessary develop, domain specific or application type specific ARCADE profiles (e.g. Healthcare ARCADE or Mobile ARCADE).

5.1 ARCADE for System Development

This section outlines a process for development of new systems using ARCADE. The overall product development process is presented in the first sub-section, while a further decomposition of the architectural description work is presented after that.

5.1.1 System Development Process

An overview of the development process is presented in Figure 5-1. The figure shows the activities performed during an enactment of the product development process to produce a particular product. The approach is very similar to the Unified Process Model [UML 1999].

There are four phases in the development process:

- **Inception**: where the scope of the project (the idea) is defined.
- **Elaboration**: where the area of concern is elaborated, features are specified, the architecture is designed and the rest of the project is planned.
- **Construction**: where the system is built in a series of increments.
- **Transition**: where the system is deployed to the user community.

The completion of a phase means that the product under development has reached a certain degree of completeness and thus represents a major milestone of the project. The milestones are shown at the bottom of the figure.

There are two kinds of activities in the development process: *process activities* and *supporting activities*. Process activities are activities that are directly related to the system development tasks. They are business analysis, architectural design, detailed design, implementation and

test. Project management and work product management are activities supporting the process activities.

The project management activity includes planning of milestones and iterations. The iteration plan should be risk driven, dealing with the requirements that involves the greatest risk in the first iterations. The work product management activity concerns the use, management and maintenance of reusable assets. It includes assistance in reuse of existing assets, collection of feedback on the reuse, and acceptance of new work products as reusable assets. Reusable assets can include architectural models, detailed design models, and implemented components.

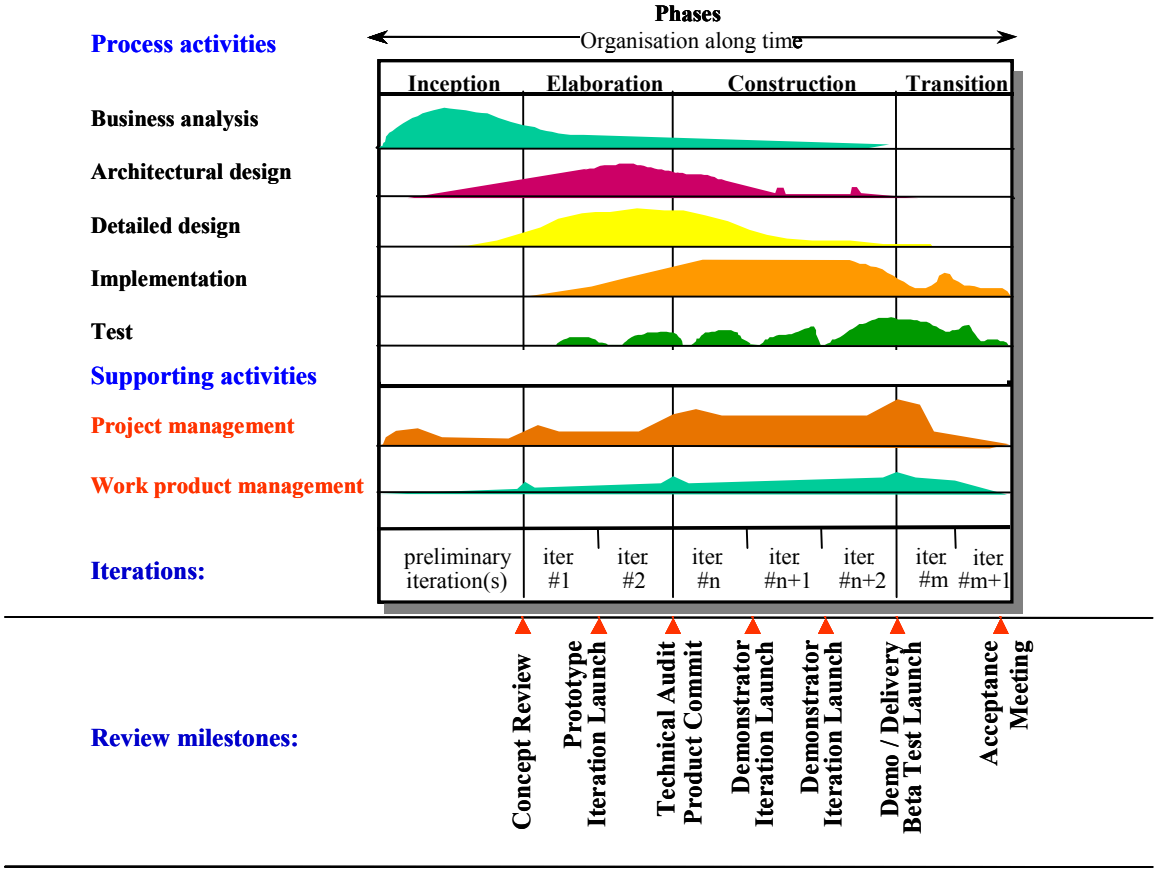

Figure 5-1 Development process for new software systems, including architectural design activity (based on [RUP])

The relative importance of the process activities varies during the life cycle of a development project. In early phases, analysis and architecture level design tend to dominate, while in later phases the majority of the work is on class level design, implementation and testing. This is illustrated in Figure 5-1, where the curves indicate the effort on each activity as a function of time.

Note that the figure shows an example. Exactly how these curves will look depends on the type of project. In more explorative projects, for instance, where the requirements and architecture is difficult to stabilise early, significant effort on requirements analysis and architectural design may persist all the way to the end of the project. In more straightforward projects, on the other hand, these activities will be more or less completed after the first two phases.

5.1.2 Iterations and Increments

The ARCADE development process follows the iterative and incremental pattern, which is now widely accepted as the preferred way of organising software development projects.

The previous most popular process pattern for development projects was the classical waterfall model. This model recommends that the software is developed in a breadth first fashion, in the sense that first all requirements are defined and agreed upon, then a design for the entire system is developed, and then the system is implemented, tested and debugged, and finally delivered to the users.

This model has serious drawbacks for product development in a dynamic environment:

- It requires that a requirement specification be frozen after the analysis phase. Thus there is little room for adjusting the course underway.
- The only possibility for user feedback before the entire application is finished is through technical documents, which works poorly.

Iterative and incremental processes have sought to overcome these problems by splitting the development into smaller steps, each ending up with an incomplete, but still usable, application that can be experimented with and improved in the next step.

Iterative process models split the development into iterations. Each iteration produces a working and testable system (part of the target system), which is experimented with and reworked and improved in the next iteration. This continues until an acceptable system emerges. This highly explorative way of developing software is suitable when little previous experience with similar systems is available. The drawback of this approach is that it is hard to plan and control.

Incremental process models split the system into components and then implement and integrate component by component. This kind of process is easier to plan and control provided that the developers have sufficient insight to get the component structure right from the beginning. This dependency on the ability to stabilise the component structure very early is also the major drawback of this approach.

The ARCADE development process is iterative and incremental. An initial statement of the context and requirements of the system is developed in the beginning. The system is then designed and implemented in steps including one or a few functionality requirements at a time (e.g. documented through use cases in the target system interface model). Each step shall result in a usable system, which can be tested and exposed to potential users. However, there is ample room for evolving the requirements, and reworking the results from previous steps. Although the process is iterative and incremental, it is highly recommended to establish the system architecture early.

Each step is an increment in the sense that it adds functionality (from one or more functionality requirements) to the product being developed. On the other hand it should be emphasised that the partially implemented system is evaluated after each step and that the plans are revised to take this experience into account. Therefore a step also constitutes an iteration and the term iteration is used in describing the process.

A visible iteration within a phase is defined as a complete development loop ending in a release of a defined product. Each release provides an increment towards the functionality the final product under development. This is depicted in Figure 5-2.

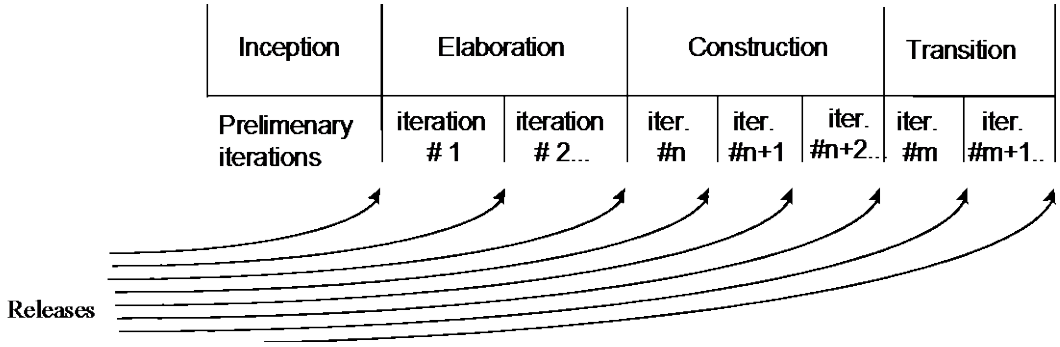

Figure 5-2 Iterations in the software development process (based on [RUP])

5.1.3 Architectural Description Development Process

The architectural description for the target system is developed in the architectural design activity of the system development process. The development of the architectural description can be seen as a sub-process of the system development process with its own internal decomposition into activities. Figure 5-3 illustrates the activities associated with each view in the architectural description, and the level of activity in each along the timeline. As for the system development process, the figure is an example, and the exact curve for each activity depends on the system being developed.

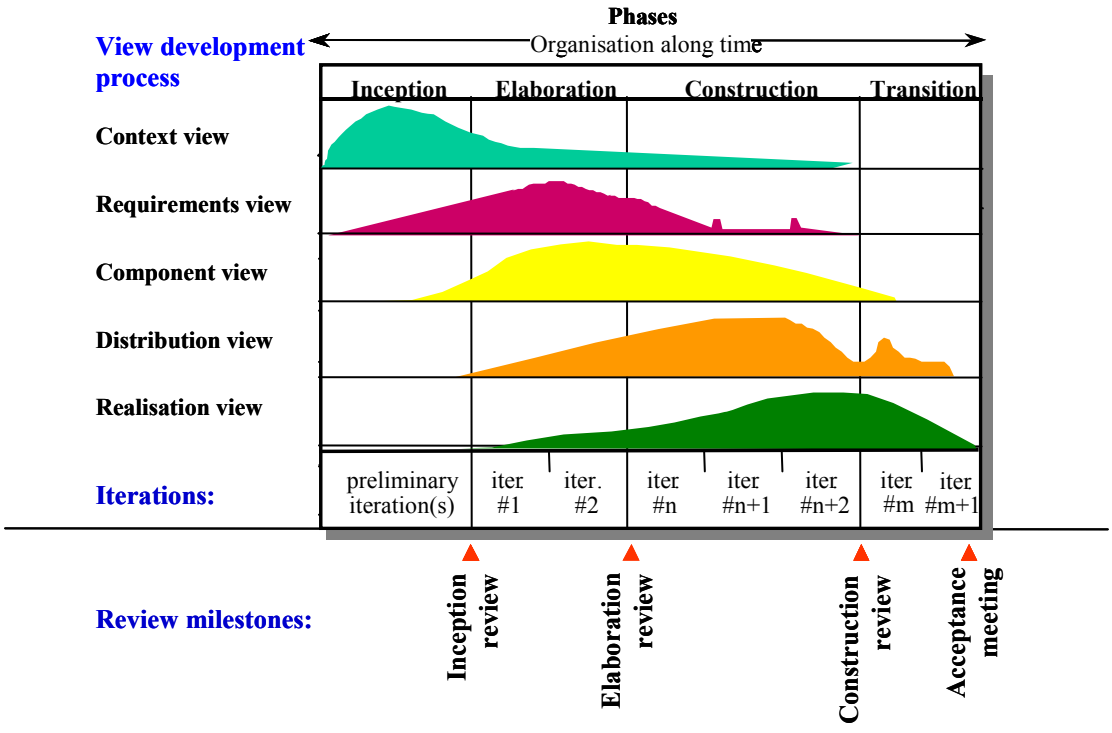

Figure 5-3 Development process for architectural description of new software systems (based on [RUP])

The four phases presented in the figure will in the context of the development of the architectural description be:

- **Inception**: where the scope and purpose of the target system are defined.
- **Elaboration**: where the area of concern is elaborated, main features are specified, the overall architecture is designed.
- **Construction**: where the system and its architectural components are described in detail.
- **Transition**: where the hand-over of the system specification to the vendor are carried out and the contract concerning the realization of the system specification are agreed and signed.

These phases do not align directly to the corresponding phases of the system development process, but some of the iterations and milestones of this sub-process are aligned with the iterations and milestones of the main system development process.

5.2 ARCADE for Documentation of Existing Systems

This section briefly discusses the use of ARCADE for documenting existing systems, and how this use differs from use in development of new systems. The context of use of ARCADE and corresponding main process differs in the two cases.

The purpose of developing and architectural description for an existing system using ARCADE may be to:

- Assist maintenance and future extensions of the system
- Assist integration into another system
- Learn from the architecture of an existing system

The architectural description process, when documenting an existing system, will usually be different than development of a new system. Activities will still be needed for each of the views as in Figure 5-3, but the timeline and weight on the different activities will depend on what kind of documentation that already exists for the system, the availability of other knowledge about the system, etc. In some cases, a reverse engineering approach will be needed, where component, requirements and context view will have to be based on a reconstruction of a realisation view from studies of the implementation.

6 Bibliography

[Buschmann 1996]	"Pattern-Oriented Software Architecture. A System of Patterns". Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michal Stal. John Wiley & Sons, Ltd, 1996. ISBN: 0-471-95869-7.
[COMBINE 2002]	"COMponent-Based INteroperable Enterprise System Development: D13 – COMBINE Development Process – Part 1 – Overview". EU Project Number IST-1999-20893. 2002.
[Gamma 1995]	"Design Patterns. Elements of Reusable Object-Oriented Software". Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison Wesley, 1995. ISBN: 0-201-63361-2.
[Gruber 1993]	"A translation approach to portable ontology specifications." T.R. Gruber. Knowledge Acquisition, 5(2):199-220, 1993.
[IEEE 1471-2000]	"IEEE Recommended Practice for Architectural Description of Software-Intensive Systems", IEEE Std 1471-2000.
[RM-ODP 1995]	"Basic Reference Model of Open Distributed Processing – Part 1: Overview and guide to use the Reference Model" ITU-TS, Rec. X901 (ISO/IEC 10746-1), Standard 1995
[RUP]	Rational Unified Process. http://www.ibm.com/software/awdtools/rup/
[Schmidt 2000]	"Pattern-Oriented Software Architecture. Patterns for Concurrent and Networked Objects". Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. John Wiley & Sons, 2000. ISBN: 0-471-60695-2.
[Szyperski 1998]	"Component Software: Beyond Object Oriented Programming". Clemens Szyperski, ACM Press, Addison Wesley Longman, 1998. ISBN: 0-201-17888-5.
[UML 1999]	"The Unified Software Development Process", Ivar Jacobson, Grady Booch, James Rumbaugh, Addison Wesley Longman, Inc., January 1999, ISBN 0-201-57169-2.

About this handbook

ARCADE is a domain and technology independent architectural description framework for software systems.

This handbook describes how to document a software architecture for a new or existing system. The framework defines a set of viewpoints from which the architecture should be described, including the context in which the systems will be used, architectural requirements, component and information model description, logical distribution, and mapping to realisation. Further, it assists the developer in defining the structure and content of the architectural description, and how to address quality concerns.

The handbook also describes how the ARCADE framework can be extended with domain and application type specific extensions.

The ARCADE framework is based on the IEEE 1471 recommend practice for architectural description of software intensive systems.