



Information Technology for European Advancement

Task 1.8 Requirements Engineering method definition (D.1.8.2b)

Version 01 - public
Edited by Loek Bokhorst

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and services**

ITEA

1. INTRODUCTION.....	3
1.1 Purpose and scope.....	4
1.2 Glossary	4
1.3 References	5
1.4 The requirements engineering process.....	6
1.5 Gathering the requirements.	7
1.5.1 Sources of requirements.....	7
1.5.2 Defining the stakeholders.	7
1.5.3 Defining the work context.	8
1.5.4 Modelling.....	10
1.5.5 Configuration	10
1.6 The product requirement specification form (SHELL).	10
2. THE ARCHITECTURE.	13
2.1 Meta architecture	13
2.2 System architecture.....	14
2.3 Software architecture.....	16
3. APPROACH.	17

1. INTRODUCTION.

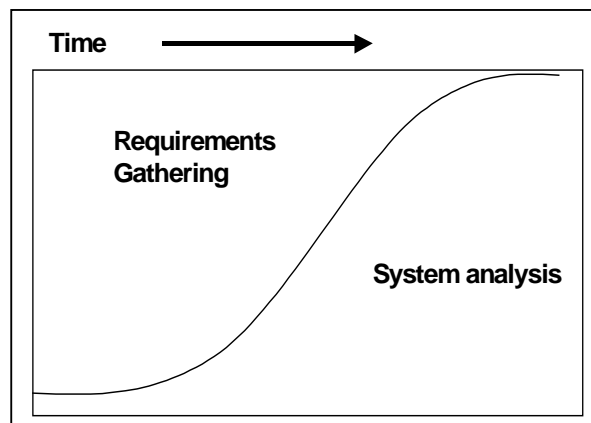
Requirements' engineering is concerned with establishing a common understanding of the requirements to be addressed by the software product. It consists of a set of transformations that attempt to understand the exact needs of a software-intensive system and convert the statement of the needs into a complete and unambiguous description of the requirements, documented according to a specified standard. This area includes knowledge of the requirements activities of elicitation, analysis, and specification.

<http://www.esi.es/Help/dictionary.html>

Requirements engineering goal is to define requirements of a product by documenting, analysing and design and implement the desired product.

The gathering and analysis of the requirements is based on the book "Mastering the Requirements Process" by James and Suzanne Robertson. It describes the way of capturing the user's requirements.

The gathering and analysis is done in parallel.



A form is used in which the aspects of a requirement is documented, like the unique identification, the source of the requirement, the stakeholder(s), the requirements type, the related Business event and product use cases etc. is documented. Refer to: The product requirement specification form (SHELL). On page: 10

The requirement is put into a Requirement Specification.

The analysis of the requirement will result into a complete description of the requirement. The modelling process of the requirement is started as early as possible. The goal of the modelling is to discover requirements of the product, learn about the product's functionality, proof that the requirement is correct, and can be designed.

The analysis phase will deliver input to the design phase where the UML diagrams are created using the Dataflow diagrams created in the analysis phase.

The architecture MetaModel is introduced being a template for the documentation of the architectural design. Refer to: THE ARCHITECTURE. On page 13

The reason for having a textual presentation of the requirement is that there are simply not very many customers who can read the pictures!

The reason for using DFD is that the requirement is analysed in the data model context, the process model context and the behavioural model context when applicable.

1.1 Purpose and scope.

This document contains the method for eliciting requirements, up to the modelling phase.

1.2 Glossary

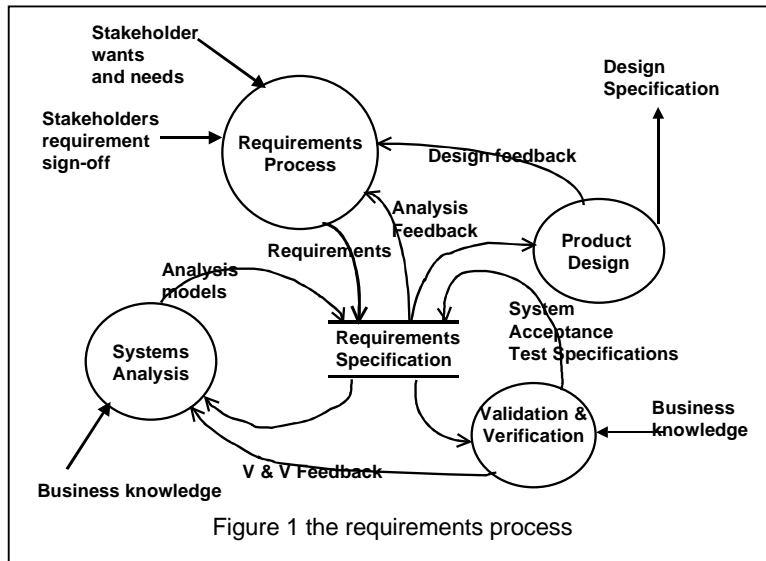
Term	Description
HW-HLD	Hardware High-level design
Meta Model	The architectural Meta Model was developed within <partner 1> (starting in 1996).
SW-ATS	Software Acceptance test specification
SW-HLD	Software High-level design
SW-ITS	Software Integration test specification
SW-UTS	Software Unit test specification
SYS-ATS	System Acceptance test specification
SYS-ITS	System Integration test specification

Term	Description
CMP	Configuration Management Plan
CR	Change Request
CRS	Commercial Requirement Specification
FRS	Functional Requirements Specification
HSI	Hardware Software Interface
IC	Integrated Circuit
ITS	Integration Test Specification
RMP	Requirements Management Plan
SAD	Software Architectural Design
SATR	Software Acceptance Test Report
SITS	Software Integration Test Specification(in system test specification)
SPMP	Software Project Management Plan
SRS	Software Requirements Specification

1.3 References

Title, author, supplier, date and status.	Reference
[1] Mastering the Requirements Process, Suzanne Robertson, James Robertson,	ISBN 0 201 36046 2
[2] "Requirements Specification template. Draft V 01- 2000-11-22"	
[3] "Requirements Specification template description.Draft V 01- 2000-11-22"	
[4] Systems engineering coping with complexity by Richard Stevens, Peter Brooks, Ken Jackson and Stuart Arnold, Prentence Hall Europe 1998.	ISBN 01309050858
[5] Managing Software Requirements A Unified Approach by Dean Leffingwell, Don Widrig, Addison Wesley, April 2000	ISBN 0201615932
[6] Software Requirements. Practical techniques for gathering and managing requirements throughout the product development cycle by Karl E. Wiegers. Released: 09/29/1999	Microsoft Press. www.Mspress.microsoft.com
[7] Programme Management and System Engineering Capability Maturity Model. , 25 September 1998.	TTM/DLS/98/5332/V0.2
[8] IEEE Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers, 1990.	IEEE Standard 610.12-1990,
[9] WP1.3 Study of timing, memory and other resource constraints	ITEA DESS
[10] Task 1.4 – Definition of Components and Notation for Components (D.1.4.2) <i>October 2000</i>	ITEA DESS
[11] Architectural Modelling, introducing the Architecture MetaModel. Yolanda van Dinther, William Schijfs, Frank van den Berk, Kees Rijnierse, Origin Technical Automation / In-Product Software. De Run 1121, 5503 LB Veldhoven, The Netherlands	www.embeddedsoftware.nl

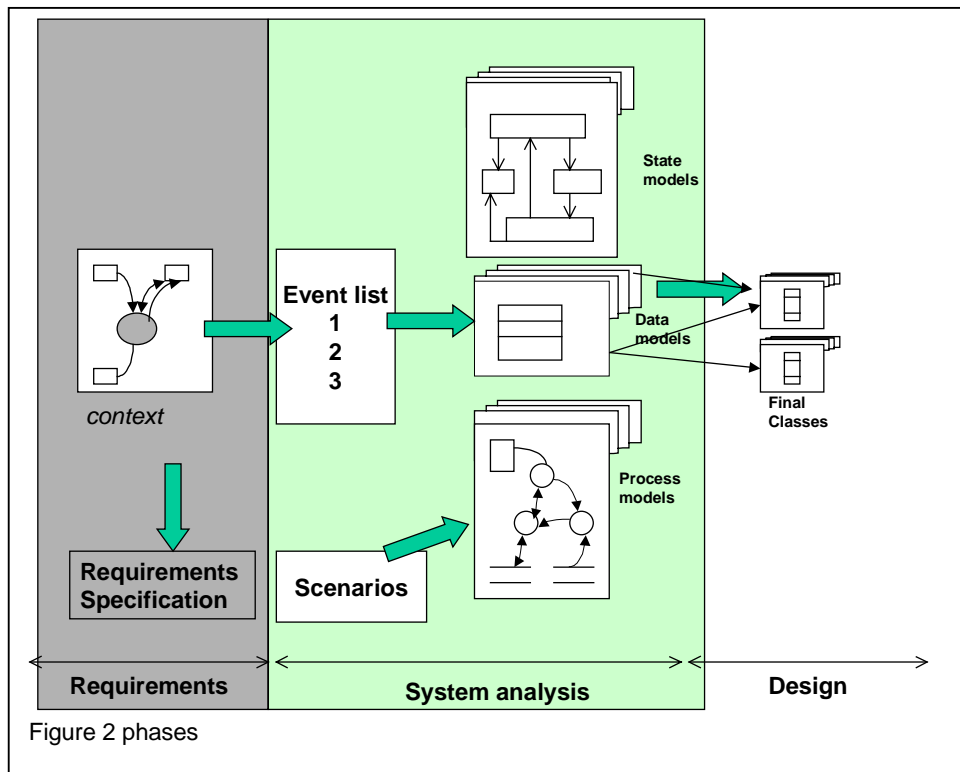
1.4 The requirements engineering process.



One or more stakeholders define the requirements of the product.

The requirement is identified and documented, analysed and finally, each requirement is signed-off by the respective stakeholders.

Parallel to the Requirements process, the system analysis process create the Analysis models and the Validation and Verification process starts creating the System acceptance test specifications. The requirements are “seen” from the different views, resulting in an early stabilisation of the requirements.



1.5 Gathering the requirements.

1.5.1 Sources of requirements.

The input for the requirements is derived from the:

1. Knowledge of the domain context.
2. Desires and ideas of the market, marketing department, commercial department, product engineers etc.
3. Existing products.
4. World standards, company standards.
5. Any stakeholder, who has a relation with the product concerned.

1.5.2 Defining the stakeholders.

The stakeholder is any person or institute, which is affected by the product or someone whose knowledge is needed to build the product.

A stakeholder can be the customer, the user, the Client, the management, the developer, the consultant, the industry, service department, training department etc.

A complete list of stakeholders is to be defined. Each stakeholder has a clear interest with one or more requirements. Their importance is defined. A stakeholder can be e.g. the holder of an essential requirement, is the major budget provider.

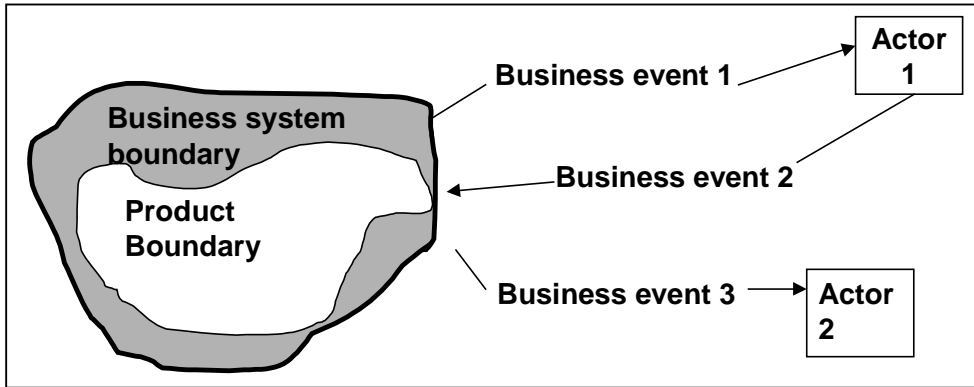
For each requirement one or more stakeholders is identified.

1.5.3 Defining the work context.

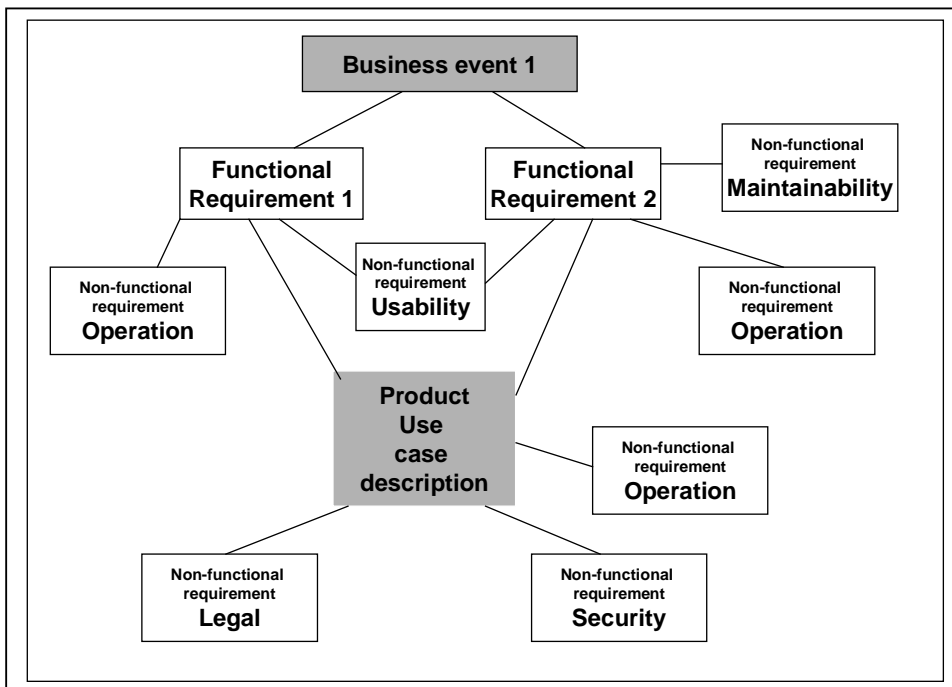
The product to be developed is part of a higher-level system or sub-system.

The context described gives a broader view of the application of the product.

By defining this scope, the actors and their events the input for the product use cases and the product scope.



The Business events will be input to the use cases and identifying the product scope.



ITEA

Define the Business event list/use cases.

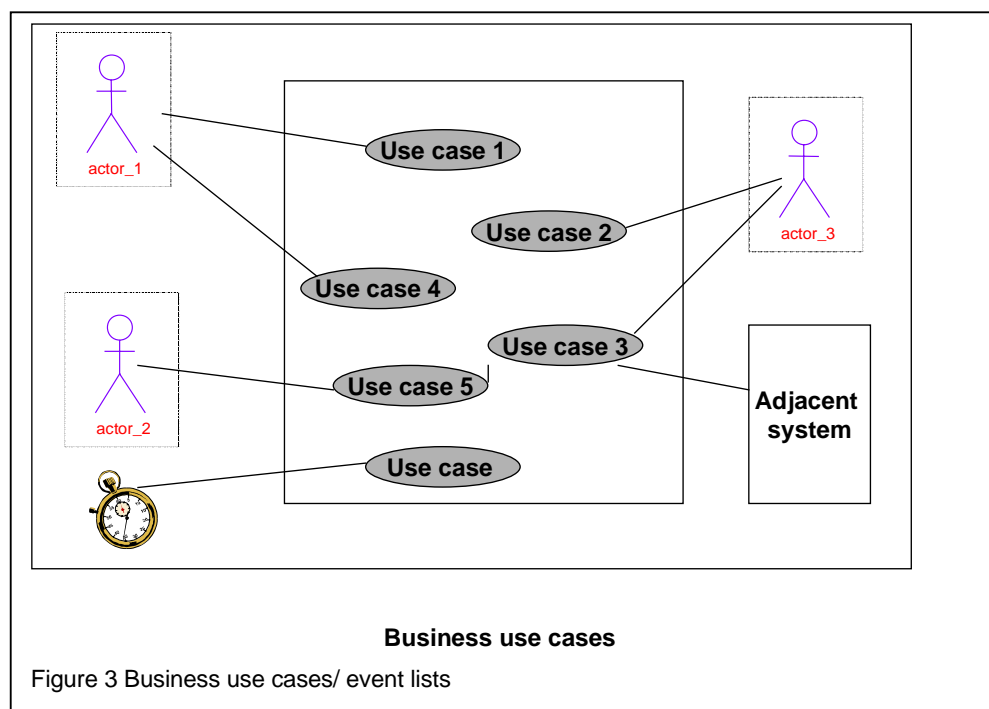
Table 1 Business event list.

Event name	Input & output
Event 1	Parm 1 (In), param 2 (out)

The event list is defined by studying the actors and the adjacent systems of the business and the inter-actions between both.

Temporal events occur when it is time to take a pre-defined business action.

A planned response to these events is described in the related use case.



1.5.4 Modelling.

By modelling the product in an early stage of the project new requirements may be discovered, knowledge of the domain is obtained, the prove of the correctness of the functionality of the requirements.

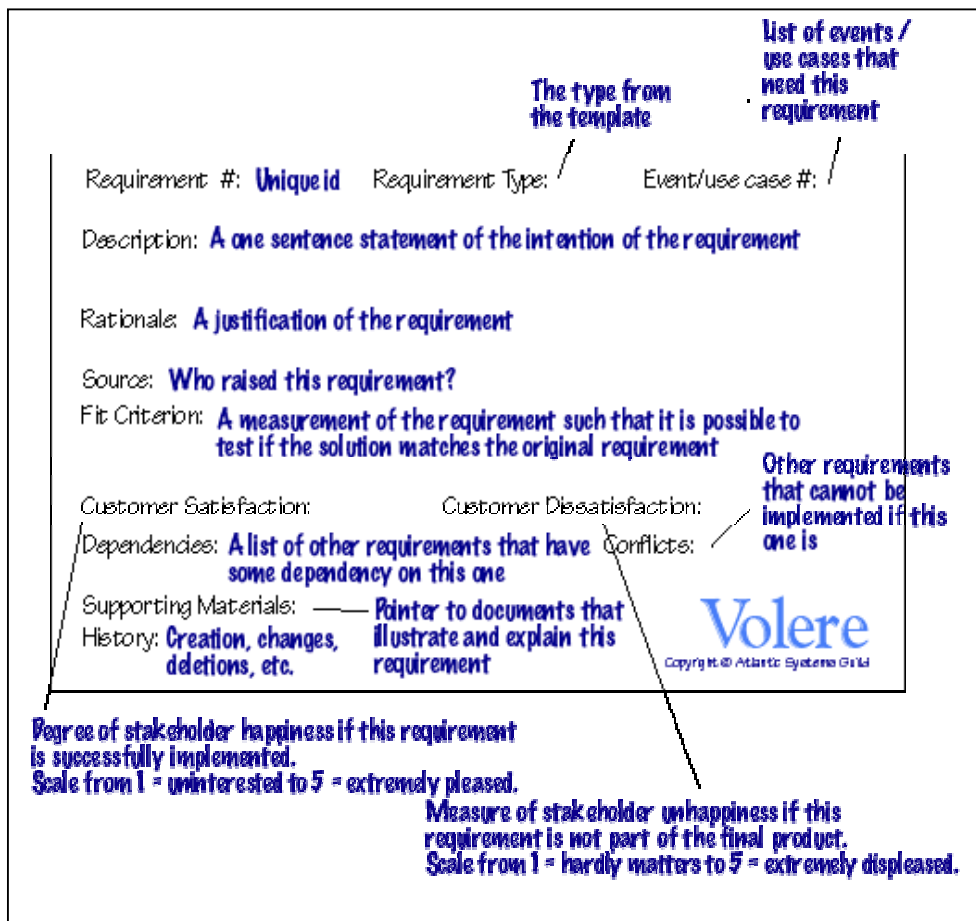
The modelling is done parallel to the requirements definition phase. Part or perhaps complete data-, process- and state- models are filled in.

1.5.5 Configuration

A short list of items which must be configurable for the library user, either compile time or run-time.

1.6 The product requirement specification form (SHELL).

Here the proposed requirement template of the Volere requirement SHELL is given as an example.



copyright © 1996 - 2000 Atlantic Systems Guild.

SHELL explanation.

Requirement Numbering

Give each requirement a unique identifier to make it traceable throughout the development process. The numbering scheme suggested in the requirement shell is:

Requirement # is the next unique requirement number

Requirement Type is the section number from the template for this type of requirement

The inclusion of the section number is not absolutely necessary because we do have a unique requirement id. However it serves as a reminder of what this requirement relates to and helps to remind why the requirement is considered important. Also the ability to compare requirements of the same type makes it easier to identify contradictions and duplications.

For example:

A functional requirement is section 9, and the next unique number is 128.

Requirement #: 128 Requirement Type: 9

We shall record the time when we are notified of a gritter truck breakdown

A performance requirement comes from section 12, and the next unique number is 129.

Requirement #: 129 Requirement Type: 12

Gritter truck drivers shall be informed of their schedule 30 minutes before leaving the depot.

Event/use case #

Is the identifier of a business event or use case that contains this requirement. There might be several Event/use case #'s for one requirement because the same requirement might relate to a number of events. The terms event and use case are already widely used in the systems development world.

We use the term business event to mean a business related happening that causes an event-response within the work that we are studying.

We use the term event-driven use case (or product use case) to mean a user-defined (or actor defined) piece of activity within the context of the product. Business events and product use cases provide a way of grouping business-related requirements and tracing them through into implementation; they are used throughout the Volere development process.

Customer Value

Customer Value is a measure of how much your client cares about each requirement.

Ask your stakeholders to grade each requirement for Customer Satisfaction on a scale from 1 to 5 where 1 means mild interest if this requirement is satisfactorily implemented, and 5 means they will be very happy if this requirement is satisfactorily implemented

The stakeholders also grade each requirement for Customer Dissatisfaction on a scale from 1 to 5 where 1 means that it hardly matters, and 5 means that they will be extremely displeased if this requirement is not satisfactorily implemented

The point of having a satisfaction and a dissatisfaction rating is that it guides your clients to think of the requirement from two different perspectives, and helps you to uncover what they care about most deeply.

Dependencies

This keeps track of other requirements that have an impact on this requirement.

If the dependency exists because requirements use the same information, then use of standard naming conventions and definitions (see Section 5) will implement this dependency.

Other dependencies exist because a solution to this requirement has a positive or negative effect on solutions to other requirements. Capture these types of dependencies by cross-referencing the requirements.

Some requirements, especially project drivers and project constraints, have an impact on all the other requirements.

Conflicts

This keeps track of other requirements that disagree with this one. Conflicts that are caused by mistake are solved simply by bringing them to the surface and resolving them. Other conflicts are because of true differences in opinion/intention. These are the conflicts that might eventually need to be addressed using negotiation or mediation techniques. There is nothing wrong with having conflicting requirements providing you know that you have them. Then you are in a position to address the conflict.

History

We follow the requirement from the date that it was created, through all its changes. We minimise future confusion by recording the rationale for making major changes. When a requirement is deleted we record when and the rationale behind the deletion. The date that the requirement passes its quality checks, and who passed it, is also recorded.

ITEA

Req. attributes	Description
Req. nr	A unique identification
Req. type	
Event/use case list	List of events/use cases that need this requirement.
Description	One sentence statement of the intention of the requirement.
Rationale	The justification of the requirement
Source	Stakeholders list
Fit Criterion	A measurement of the requirement: What test proves the correctness of the solution.
Customer Satisfaction	Value: 1 - 5
Customer Dissatisfaction	Value: 1 - 5
Req. dependencies	
Req. conflicts	
Supporting material	
History	

2. THE ARCHITECTURE.

Architectural modelling is a technique to generically structure architectures into various complementary units. Each unit addresses a specific concern, thus enabling separation of concerns. Architectural modeling also helps to decrease complexity, to ensure completeness of the architecture, to organize its documentation and to improve communication with the various stakeholders.

In literature, a few architecture models have been published. Two examples are the “Soni model” [1] and the “4+1 View Model” [2], both applicable to software architecture.

Here an architecture reference model is presented that is applicable to various kinds of architecture, hence its name: the Architecture MetaModel. The model describes both a generic Meta architecture and specific versions for both system and software architecture. A 6-view approach is used, which is based on three levels of architectural description. For each level, a static and a dynamic view are presented. The static view describes the components of the level whereas the dynamic view describes the interaction between these components.

2.1 Meta architecture

The Meta architecture specifies three levels of abstraction:

1. ***Conceptual level:*** This is the highest abstraction level. Here the architecture is described in terms of black box reasoning. This means that the context of the architecture and the essential aspects from the requirements specification focusing on non-functional requirements are taken into account. No design decisions are presented yet, only an analysis of the application area is presented including long-term rules that will apply to the future architecture.
2. ***Logical level:*** Here all design decisions are described and their rationales explained (white box reasoning). The architecture is defined in terms of decomposition, a set of design rules and the dynamic aspects of the design.
3. ***Physical level:*** Typically, an architecture is not a product of its own. It is input to other activities that add more detail. For the system architecture, these activities are the development of the mechanical, electrical and software architecture. For the software architecture, the activity is the software implementation.

Whereas the logical level concentrates on the design and its behaviour, the physical level is necessary to enable the transition from the design to the detailed activities. It defines a mapping and a set of rules and conditions.

Each level is partitioned into two parts:

1. ***Static:*** The static part describes the components and concepts that are important for the level. In case of the logical level, the design can incorporate multiple nested levels of component definition.
2. ***Dynamic:*** The dynamic part describes the interaction between the components, the run-time behaviour and examples of how the architecture will operate (e.g. use cases and scenarios). The latter is meant to gain insight and to verify correctness of the architecture.

The following figure gives an overview:

Conceptual:

Black box view of the system describing its context and the user's view.

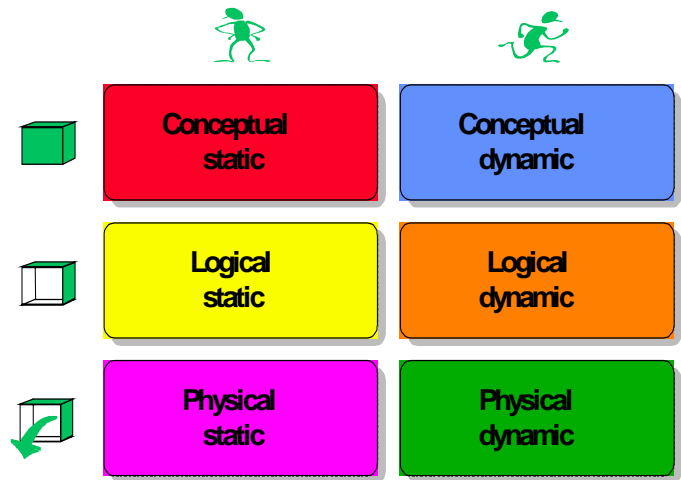
Logical:

White box view of the system describing the actual (top level) design.

Physical:

Transformational view of the system describing the consequences of the design for the implementation.

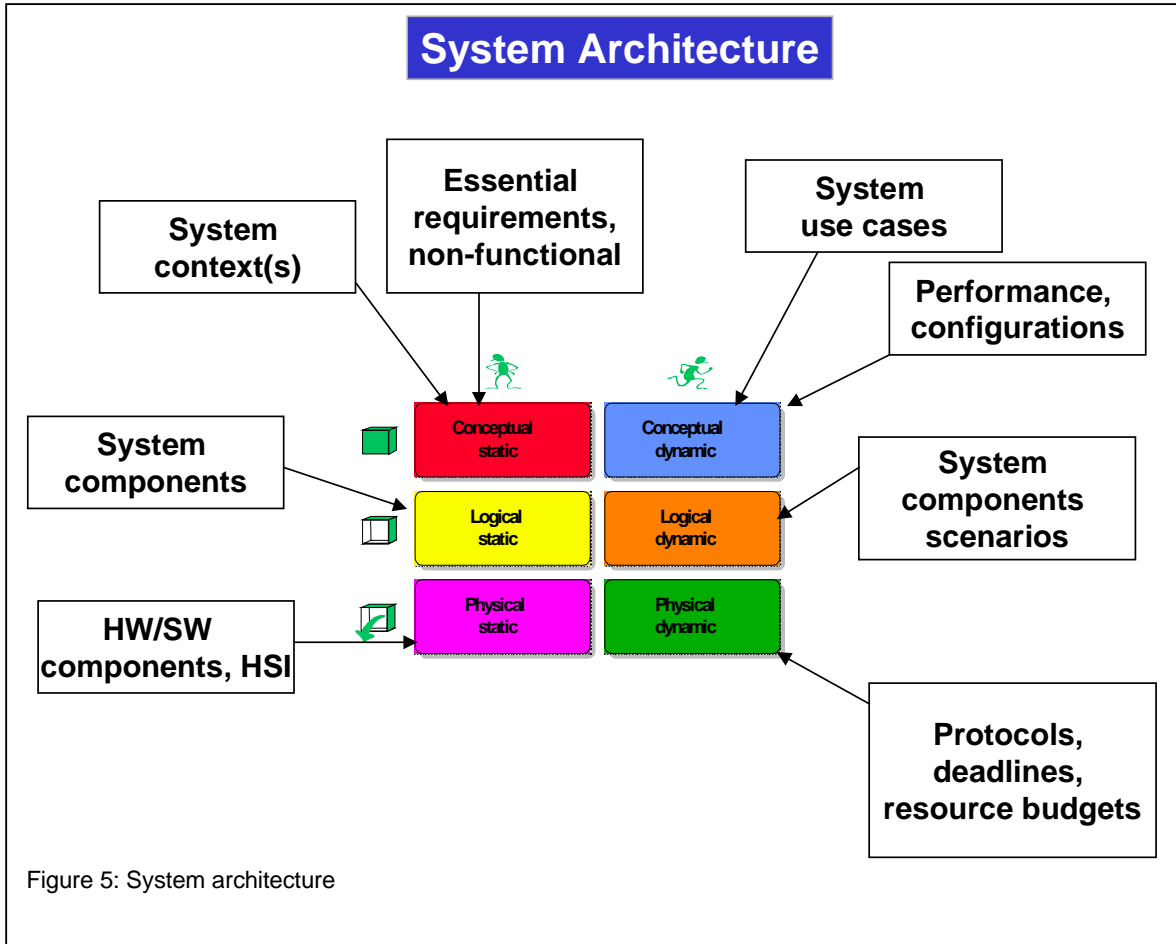
The dynamic view depends on the static view.



2.2 System architecture

When adopting the Meta architecture for the system architecture, the following six Views emerge:

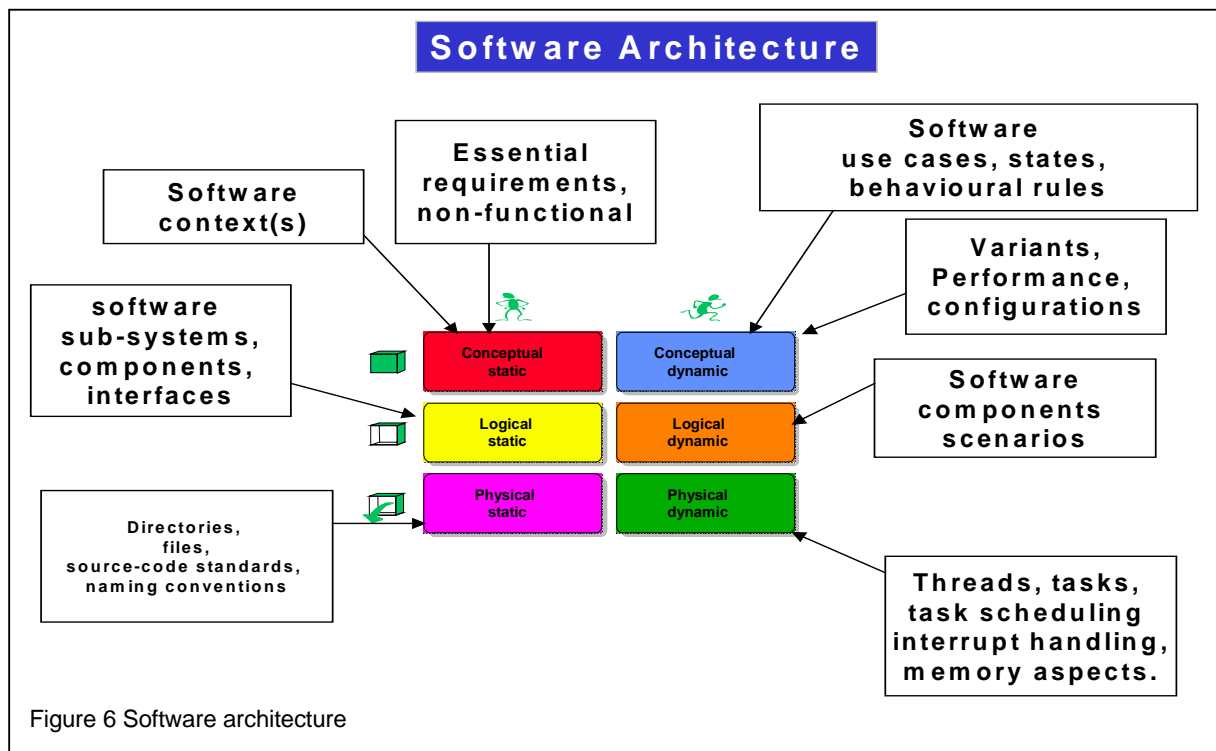
1. **Conceptual static:** This View describes the system context, types of users, fundamental and non-functional requirements. These are described in terms of actual and future context and requirements.
2. **Conceptual dynamic:** This View describes typical usage (e.g. by means of system use cases) and behavioural rules. Also, various system configurations and variants and how they are switched are described. Performance requirements are defined here as well.
3. **Logical static:** This View describes the system in terms of system components and their mutual dependencies. These system components are defined independently of the actual hardware and software. Design paradigms can be defined that are to be maintained throughout the system (both with an actual and a future scope).
4. **Logical dynamic:** This View contains the dynamic behaviour of the system components, e.g. how and when a component is connected or disconnected, what kind of communication takes place. System start-up and shutdown rules are to be defined and examples of internal system operation can be added (e.g. scenarios).
5. **Physical static:** This View transfers each system component and the connections between them into a set of mechanical, electrical and software parts. For each part, specifications and interfaces are defined (e.g. mechanical dimensions of electrical components).
6. **Physical dynamic:** This View adds to the physical static View a number of dynamic aspects like protocols and run-time resource budgets (memory and performance) for all mechanical, electrical and software parts.



2.3 Software architecture

When adopting the Meta architecture for the software architecture, the following six Views emerge:

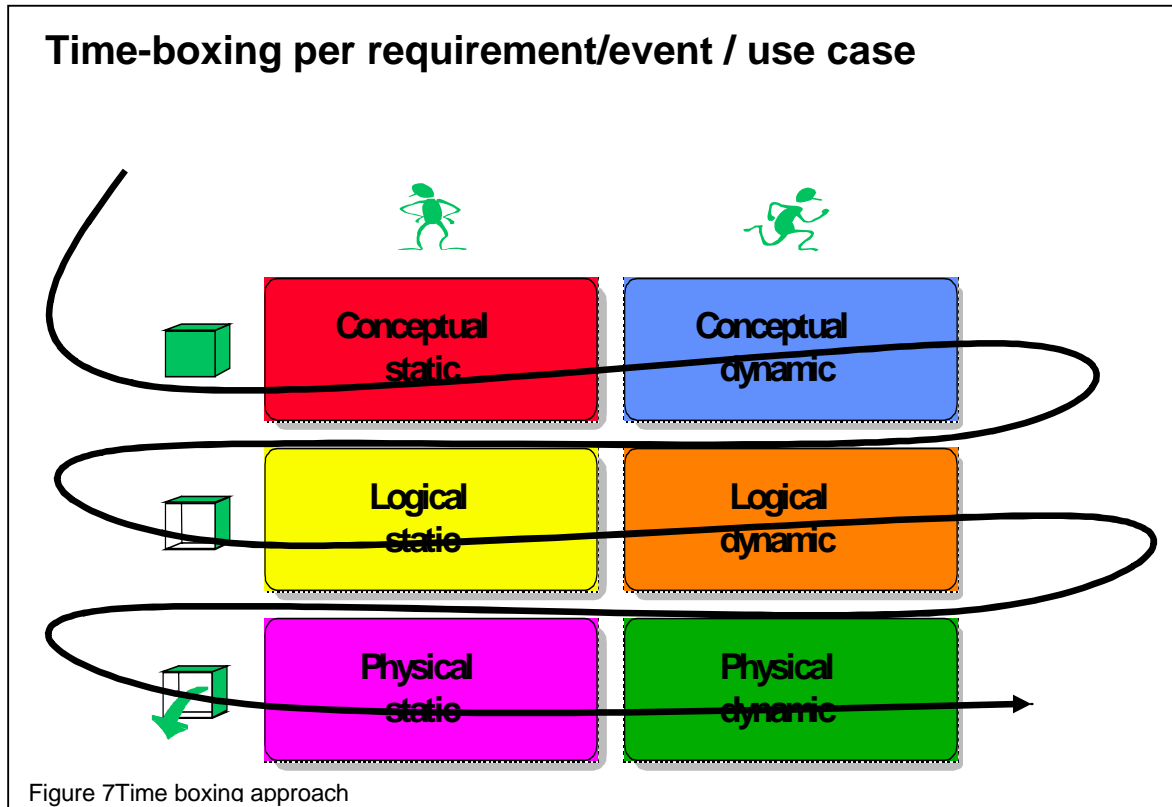
1. **Conceptual static:** This View describes the context of the software architecture including a context diagram, external interfaces and resources such as libraries and operating systems, fundamental software requirements and possible future extensions.
2. **Conceptual dynamic:** This View describes typical usage (software use cases), user-visible software states, configurations and variants, software performance requirements and software behavioural rules.
3. **Logical static:** This View is the heart of the software architecture in that it defines both the basic paradigms of the software (e.g. via design patterns) and the decomposition of the software. The decomposition takes place in subsystems and/or a layering model. The actual decomposition is dependent on the design method chosen. Also, the major internal interfaces in the software system need to be defined and all persistent data aspects are to be described.
4. **Logical dynamic:** This View describes all design-related run-time aspects, including mechanism descriptions, start-up and shutdown behaviour, major algorithms and inherent parallelism. Also, examples of internal software operation can be added (scenarios) to verify correctness of the design.
5. **Physical static:** This View focuses on the file, directory and code level. This means that the design needs to be mapped to a file and directory structure. Also, rules for constructing libraries and executables are included as well as coding and code naming standards.
6. **Physical dynamic:** This View focuses on run-time processes, threads and tasks (defining generic rules and a physical mapping of the design). Also, scheduling, interrupt aspects and performance and memory issues are to be considered here.



3. APPROACH.

The architectural views are filled in with the product in mind, the individual requirement, events/use cases described in the system analysis phase. All the applicable views are addressed such that first the conceptual aspects are filled in, followed by the logical aspects and the physical aspects. When the work cannot be completed fully e.g. there is no more inspiration, or when the level of abstraction defined for iteration is reached the next view is addressed. In this way the architectural model description is kept consistent.

A number of iterations may be necessary to complete the processing. This approach is called time-boxing.



Depending on the size of the project one or more persons are allocated to a specific view, or are responsible for one or more views.

APPENDIX 1 THE TEMPLATE OF THE ARCHITECTURE META MODEL.

APPENDIX 1.1 Conceptual static view

Conceptual static	Description
-------------------	-------------

ITEA

view	
Context	This section contains one or more context diagrams describing the surroundings of the system, depicting the system itself as a black box. This can be done in terms of: Surrounding hardware, surrounding software, surrounding files the end-user (possibly different types of user).
External resources	This section actually lists the various external resources that need to be available for the software system. For each of the items listed, a short explanation and an indication of the delivering party are available. Typical items to be listed are: surrounding hardware, software libraries (that are delivered to be linked into the software product), software executables (to communicate with), resource files (e.g. bitmaps, fonts, user-defined files to be incorporated), standards (e.g. protocols).
Typical users/actors	This section describes the types of user/actors that will be using the software system. This list of users is mainly input to the use cases.
Major user-visible functions	This section describes the major user-visible functions (not software functions but functionality delivered by the software). Also, describe the relation between the user-visible functions and the users (which user is allowed to perform a function).
External interfaces	This section describes the external programming interfaces that are delivered by the software system, i.e. API's.
Axioms	This section describes the main consequences from the requirements specification that will be essential to the design. Focus on the aspects of the requirements that have a high impact on the logical view.
Future extensions	This section describes a number of possible future extensions to the system. Also, give an indication of how difficult it would be to introduce this extension.

APPENDIX 1.2 Conceptual dynamic view

Conceptual dynamic view	Description
Typical usage	This section contains a number of typical examples of how the software system will be used. This can be described by means of use cases, pictures, lists or plain text.
User-visible software states	This section describes the actual states that do show to the user. As a technique, typically use a state transition diagram and indicate what the user notices in each of the states.
Versioning	This section describes the various ways of delivering the software system. Versioning describes the variants that emerge when switching options/features on and off or by having country specific settings. This section describes all possible combinations and how they are dealt with (externally to the design). Typical issues here can be static and dynamic upgrading, interoperability of different versions and the mechanisms for switching between options/configurations.
Performance requirements	This section can be a brief summary of the requirements specification's performance statements. As performance issues are often difficult to predict, extra information should be available about feasibility, importance, and consequences for the design and hardware relations.
Rules & guidelines	This section can be used to describe typical dynamic rules of the software

ITEA

Conceptual dynamic view	Description
	system's behaviour.

APPENDIX 1.3 Logical static view

Logical static view	Description
Paradigms	This section describes design concepts that are used as starting points for this design. Typical examples could be the design patterns as described in the book by Gamma et al. The paradigms section should be viewed as a basis for extension and reuse. It should help in determining how new functionality could fit into the design.
Subsystems	This section describes some decomposition of the system. This can be functionality based (e.g. SASD) and/or object based (e.g. UML, OMT). The decomposition itself can be described in more than one picture.
Layering model	This section should be based on the subsystem section. It describes the layering model, i.e. a hierarchical model that defines the decomposition of the system adding general calling conventions. Next to a picture, this section must describe the functionality of each layer in text.
Dependencies	This section describes the type of the relations between the elements of the layering model in more detail (e.g. calls, call-backs, subscription, and inter-process communication).
Major internal interfaces	This section describes selectively the major internal interfaces within the system.
Persistent data	This section defines the stored data model of the software system or any other type of persistent storage.
Rationale	This section describes the reasons of the design choices made in the previous sections. Main goal is to document history and to prevent the same discussions from starting again.

APPENDIX 1.4 The logical dynamic view.

Logical dynamic view	Description
Scenario's	This section describes typical examples of how the software will co-operate. This can be done by means of message sequence charts or calling trees. Use this section to get a better insight in the functionality of the software and to validate the logical static view.
Internal SW states	This section can be used to describe the states of the software or parts of the software. Use state transition diagrams as a technique and make sure to describe transitions in data and/or functional changes.
Mechanisms	This section is to describe all mechanisms in the software system that are typically used by all or a lot of parts. Examples are error and exception handling, memory management, messages and events, assertions, file handling, DB access, UI tools, graceful degradation solutions.
Run-time entities	In case of using OO techniques, this section describes the run-time budgets for object entity creation and gives information on the multiplicity of the run-time relations between classes. Possible non-OO type of information can be an indication of the multiplicity of the number of mallocs for structures and arrays.
Start-up & shutdown	This section is to be used to describe typical start-up and shutdown behaviour. For start-up, the initialisation order of parts of the system should be described.

ITEA

Logical dynamic view	Description
Inherent parallelism	This section is to be used to describe inherent parallelism and synchronisation. This is an alternative for describing scenarios. Typically, an activity state diagram technique could be useful.
Major algorithms	This section can be used to describe the typical core functionality of the software.
Rationale	This section describes the reasons of the design choices made in the previous sections.

APPENDIX 1.5 Physical static view

Physical static view	Description
File & directory mapping	This section describes how the logical static view decomposition maps onto files and directories. Define also general naming conventions and file-extension rules
Libraries & executables	This section describes how files/directories form together a library or an executable. Also describe make files, compiler settings, paths where the targets are put and how the software release structure is set up.
Coding & detailed design rules	This section describes the coding rules (layout, naming conventions, safety,) and design rules (do's and don'ts of detailed designing). Often this section is a reference to a separate document.
Templates	This section defines templates for subsidiary documents, especially the detailed design document. Often this section is a reference to a number of separate documents.

APPENDIX 1.6 Physical dynamic view

Physical dynamic view	Description
Process paradigms	This section contains the general rules that describe the underlying philosophy in process creation (when to create a thread or a process and when not to). This section can be compared with the paradigm section in the logical static view. It should be viewed as a basis for extension and reuse.
Processes & threads	This section describes a model for actual processes and threads. This can be done by means of a process model (graphical notation). Also, each process/thread is to be described in terms of the software parts that run within it, the processor it is assigned to and a number of rules for the multiplicity, the priority and life cycle of the process/thread.
Scheduling & priorities	This section describes the scheduling mechanism(s), e.g. (non-) pre-emptive, round robin, time slicing, If there are different priorities to attach to processes and threads, these should be described here.
Events & interrupts	This section describes the events and interrupts in the system. Events are used as a general term for asynchronous behaviour (user/software/hardware originated), whereas interrupts are hardware originated events with a high priority.
Performance issues	This section describes what is done to deal with the performance requirements (time related)
Memory budget	This section describes what is done to deal with the memory requirements. Both foreground memory (e.g. RAM) and background memory (disk space) are to be dealt with.

ITEA

Rationale	This section describes the reasons of the design choices made in the previous sections. Main goal is to document history and to prevent the same discussions from starting again.
-----------	---

APPENDIX 1.7 Development process issues

Development process issues	Description
Development dependencies	This section describes the consequences of the chosen architecture for development order and organisation (into teams). Including the creation of stubs and emulators. Also dependency management rules can be included here.
Testing issues	This section describes how the software system is to be tested with respect to the design. (If the testability of the design turns out to be inadequate, the design should be reconsidered.) Testability includes rules for module test cases and integration testing.
Configuration management	This section describes (optionally) whether the architecture leads to specific requirements for configuration management. Consider the versioning section (0) for input.
Release management	This section describes the consequences for the releasing of the software. See section 0 for input.