Evolution Management and Process for
Real-Time Embedded Software Systems

Framework for Requirements

Deliverable 3.1

Edited by Nadine Heumesser

5 April 2004

Version 1.0

Status: Final

Public Version

ITEA
INFORMATION TECHNOLOGY
FOR EUROPEAN ADVANCEMENT

Authors/Partners:

Partner	Author	Email
BarcoView	Andy De Mets, Lieven Demeestere	Andy.demets@barco.com Lieven.demeestere@barco.com
DaimlerChrysler AG, Ulm (RIC/SP)	Nadine Heumesser, Hannes Omasreiter, Ramin Tavakoli, Frank Houdek, Joachim Weisbrod, Thomas Zink	Nadine.Heumesser@daimlerchrysler.com Hannes.Omasreiter@daimlerchrysler.com Ramin.Tavakoli_Kolagari@daimlerchrysler.com Frank.Houdek@daimlerchrysler.com Joachim.Weisbrod@daimlerchrysler.com Thomas.TZ.Zink@daimlerchrysler.com
Fraunhofer FIRST	Jens Gerlach, Matthias Tief	jens@first.fhg.de tief@first.fhg.de
Fraunhofer IESE	Isabel John, Jörg Dörr, Daniel Kerkow, Antje von Knethen, Barbara Paech, Tom König, Thomas Olsson	john@iese. Fraunhofer.de Joerg.Doerr@iese.fraunhofer.de Daniel.Kerkow@iese.fraunhofer.de Antje.vonKnethen@iese.fraunhofer.de Barbara.Paech@iese.fraunhofer.de Tom.Koenig@iese.fraunhofer.de Thomas.Olsson@iese.fraunhofer.de
HOOD-group, Munich	Hans-Dieter Maier,	Hans-Dieter.Maier@hood-group.com
Jabil Circuit	Linde Loomans	linde.loomans@philips.com

SIEMENS AG	Ricardo Jimenez	Ricardo.jimenez@mchp.siemens.de
TU München	Bernhard Deifel	deifel@in.tum.de
Validas	Oscar Slotosch, Hans-Peter Zängerl	slotosch@validas.de zaengerl@validas.de

Document History:

Date	Version	Editor	Description
5 April 2004	1.0	Stefan Van Baelen	Public version based on internal version 2.02

Filename: D3.1_v1.0_Public_Version.doc

Abstract

In this deliverable a framework for requirements is defined by studying given requirements for evolving embedded systems, classifying requirements, and allocating these requirements to the various abstraction layers, which could, for example, be a system hierarchy. Furthermore, we define processes for dealing with these classification schemes, e.g., processes for eliciting and analyzing requirements.

Keywords:

Requirements, Non-Functional Requirements, Requirements Engineering, Framework, Classification Scheme, Abstraction Layers, Models, Elicitation, Analysis, Product Lines, Product Families

Table of Contents

1 Introduction

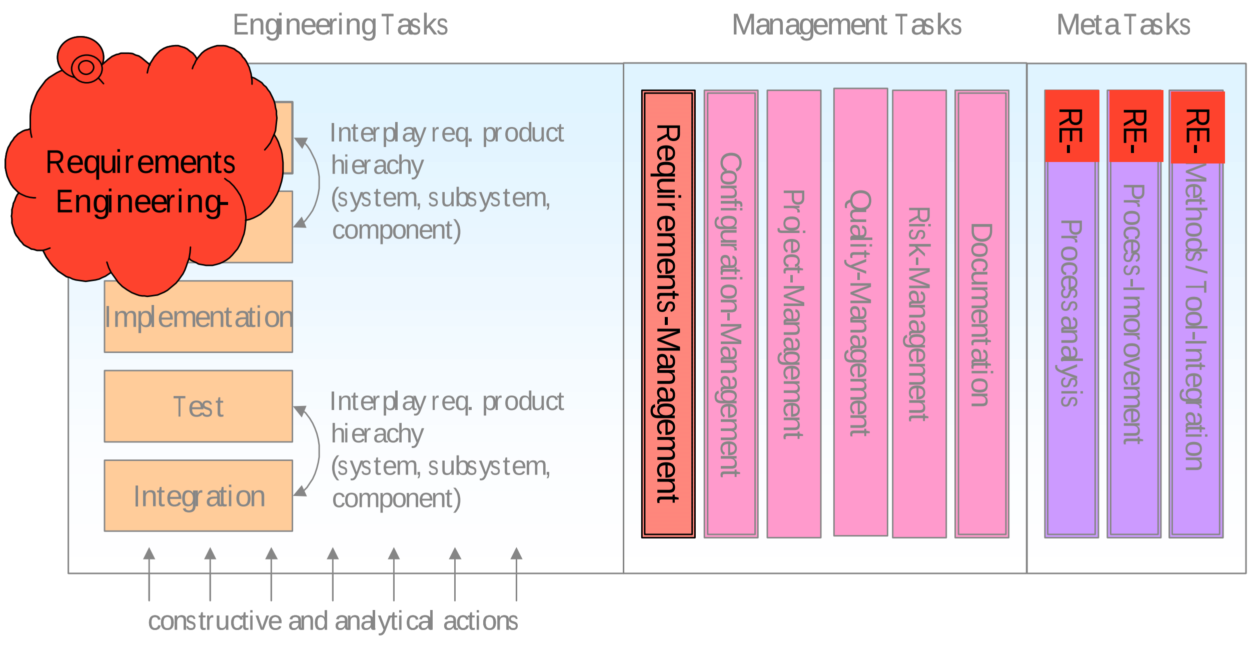


Figure 1-1 shows the distinction between Requirements Engineering and Requirements Management for the context of the EMPRESS Project

Requirements Engineering (RE) encompasses activities at the beginning of the product development cycle (assumed a non-iterative development process is followed), aiming to define the product to be developed. Typical activities for functional and nonfunctional requirements are:

- Elicitation of requirements (i.e., identify stakeholders and other sources of requirements; identify needs, requirements, and constraints from the sources; establish an agreement between various stakeholders and reconcile inconsistency and incompleteness)
- Documentation of requirements (i.e., classify and group requirements according to their contents)
- Analysis of requirements (i.e., identify inconsistencies and incompleteness; test if the set of requirements is internally consistent and complete and conforming to its source)

In this deliverable a framework for requirements is defined by studying given requirements for evolving embedded systems, classifying requirements and allocating these requirements to the various abstraction layers, which could, for example, be a system hierarchy. Furthermore, processes for dealing with these classification schemes are defined, e.g., processes for eliciting and analyzing requirements.

1.1 EMPRESS Process – Refinement of Process Discipline Requirements

The discipline Requirements of the EMPRESS-Process is divided in the two sub-disciplines Requirements Engineering (RE) and Requirements Management (RM).

This chapter focuses on the sub-discipline RE, its activities and sub-activities. In addition, it relates the approaches and methods developed as a result of EMPRESS to the RE activities and sub-activities.

A detailed description of RM is given in D.3.2.2.

In the following, we describe the structure of the sub-discipline Requirements Engineering in more detail.

First we distinguish between three activities:

- Elicitation: Focusing on gathering information/requirements
- Analysis: Dealing with building and organizing the requirements space

and

- Documentation: Recording the identified unstructured and structured requirements.

Each of these activities includes several sub-activities explained below.

Figure 1-2 gives an overview of the activities and sub-activities that are part of the sub-discipline Requirements Engineering.

(Please note that the structure gives no evidence on the chronological order of the activities.)

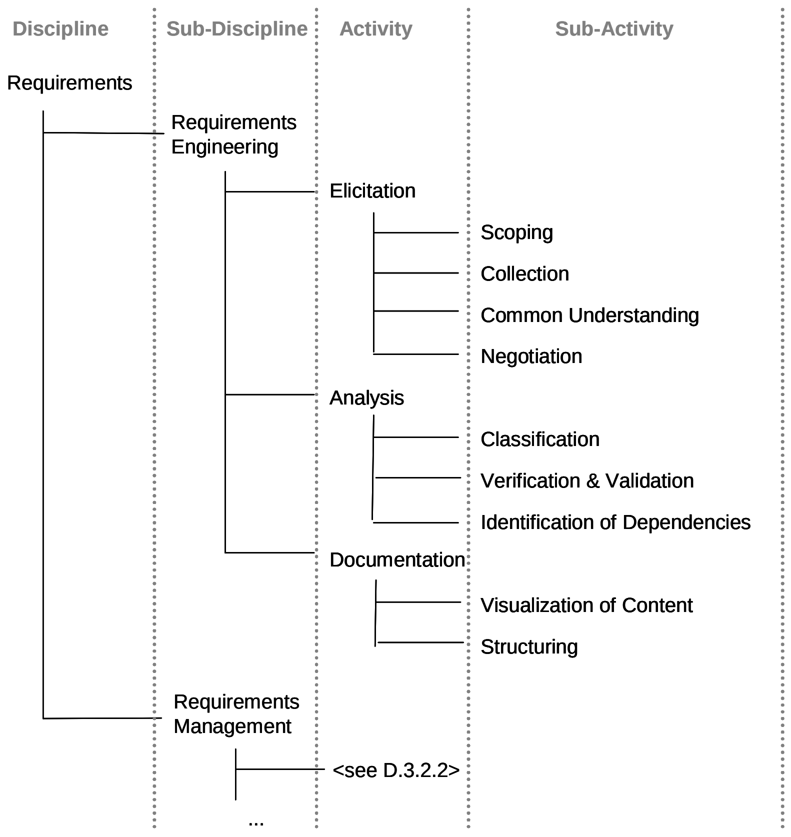


Figure 1-2: Structure of the EMPRESS-Process discipline Requirements

Scoping, Collection, Common Understanding, and Negotiation have been identified as sub-activities of the activity Elicitation.

- **Scoping:** First of all the scope of the system that has to be developed has to be marked. This means that the stakeholders have to be identified and that the context in which the system will be used has to be prepared. (Only if we know the target group, we are able to build a system satisfying their needs.)
- **Collection:** In addition to the sub-activity Scoping, an initial collection of information/

requirements by studying recent requirements documents, laws and standards, marketing material etc. has to start.

- **Common Understanding**: During the entire Elicitation activity different discussions and reviews with stakeholders have to take place. On the one hand the goal is forming a common understanding. On the other hand we want to achieve a refinement of different requirements for a better understanding.
- **Negotiation:** In this sub-activity, the feasibility has to be proved. For example, solutions and business objectives how and if the implementation of the requirements makes sense are discussed. In addition we are going to freeze the discussed set of requirements. This baseline builds the basis for the contract and consequently for further discussions based on, for example, changes in the set of requirements and their effects.

The activity Analysis is divided in the sub-activities Classification, Verification & Validation, and Identification of Dependencies.

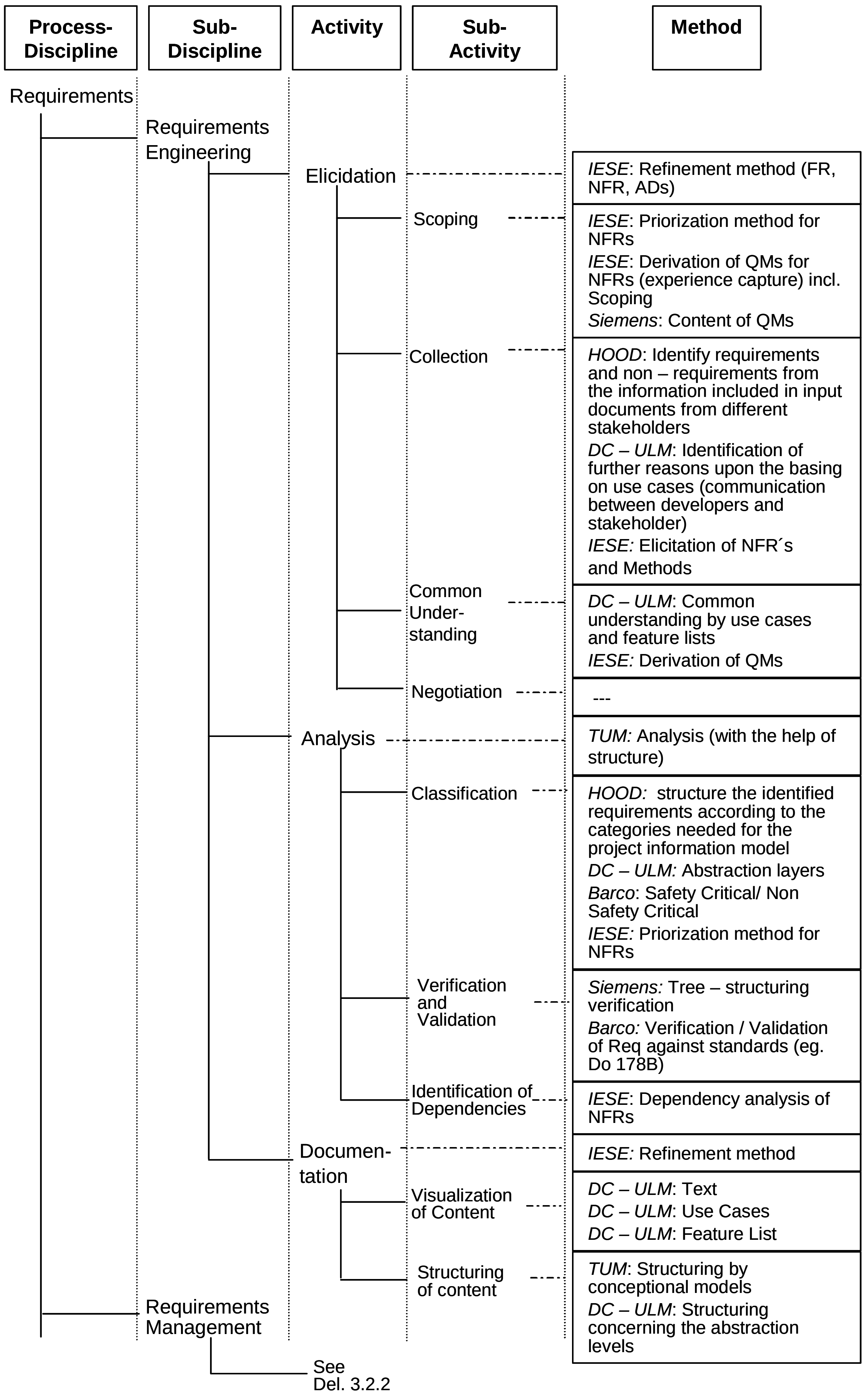
- **Classification:** During the Classification different clusters of requirements have to be built. Requirements have to be divided and conquered. For example by assigning priorities to requirements. We can group them into the ones that are most important to be implemented first and the ones that are less important. In addition the necessity of a special treatment of groups of requirements can be identified.
- **Verification & Validation:** In this context Verification means an internal check on consistency and completeness of a selected set of requirements, whereas Validation checks the compliance with underlying needs and rules, e.g., compliance with standards.
- **Identification of Dependencies:** This sub-activity deals with the identification of relations between all requirements. These relations might be very useful if changes have to be incorporated consistently in a set of (dependent) requirements or if requirements are added or deleted (which might cause follow-up changes or deletions).

The Sub-activities Visualization of Content and Structuring from the activity Documentation.

- **Visualization of Content:** During this sub-activity the requirements are written down in a chosen notation, possibly by using provided templates, databases,...
- **Structuring:** Structuring means setting up the structure for visualization. Thus it defines visualization methods for the various kinds of information and their combination.

The table below finally gives an overview of the methods developed in EMPRESS and their assignment to the different activities and sub-activities of the sub-discipline Requirements Engineering.

For a detailed description of the several methods we refer to the corresponding contributions below (see Figure 1-3).



Process-Discipline	Sub-Discipline	Activity	Sub-Activity	Method
Requirements	Requirements Engineering	Elicidation		*IESE*: Refinement method (FR, NFR, ADs)
			Scoping	*IESE*: Priorization method for NFRs *IESE*: Derivation of QMs for NFRs (experience capture) incl. Scoping *Siemens*: Content of QMs
			Collection	*HOOD*: Identify requirements and non – requirements from the information included in input documents from different stakeholders *DC – ULM*: Identification of further reasons upon the basing on use cases (communication between developers and stakeholder) *IESE:* Elicitation of NFR´s and Methods
			Common Under-standing	*DC – ULM*: Common understanding by use cases and feature lists *IESE:* Derivation of QMs
			Negotiation	---
		Analysis		*TUM:* Analysis (with the help of structure)
			Classification	*HOOD:* structure the identified requirements according to the categories needed for the project information model *DC – ULM:* Abstraction layers *Barco*: Safety Critical/ Non Safety Critical *IESE:* Priorization method for NFRs
			Verification and Validation	*Siemens:* Tree – structuring verification *Barco:* Verification / Validation of Req against standards (eg. Do 178B)
			Identification of Dependencies	*IESE*: Dependency analysis of NFRs
		Documen-tation		*IESE:* Refinement method
			Visualization of Content	*DC – ULM*: Text *DC – ULM*: Use Cases *DC – ULM*: Feature List
			Structuring of content	*TUM*: Structuring by conceptional models *DC – ULM*: Structuring concerning the abstraction levels
	Requirements Management	See Del. 3.2.2		

Figure 1-3: Assignement of the methods developed in EMPRESS to activites and sub-

activities of the sub-discipline Requirements Engineering.

The methods of the different partners mentioned above can be found in the following sections of this document:

Contribution of Partner	Section
Barco	2.1
Jabil	2.2
DC Ulm	2.3
Validas	4
TUM + TUE	5
HOOD	6.1
IESE	6.2
Siemens	6.3
FIRST	6.4

2 Classification Schemes and Abstraction Layers

2.1 Classification Scheme in the Avionics Domain

2.1.1 Introduction

BarcoView, Avionics Division develops and produces a complete range of cockpit displays, from Control and Display units, Cockpit Head Down Display to Multi-Function Displays (Primary Flight, Engine Instrument, ...). These displays provide an interface for the pilot to, in many cases critical, equipment like a mission computer, engines. They are supporting the pilot in flying the airplane. As such our displays are integrated in the complete avionics system, communicating with the distributed set of equipment/computers of the aircraft.

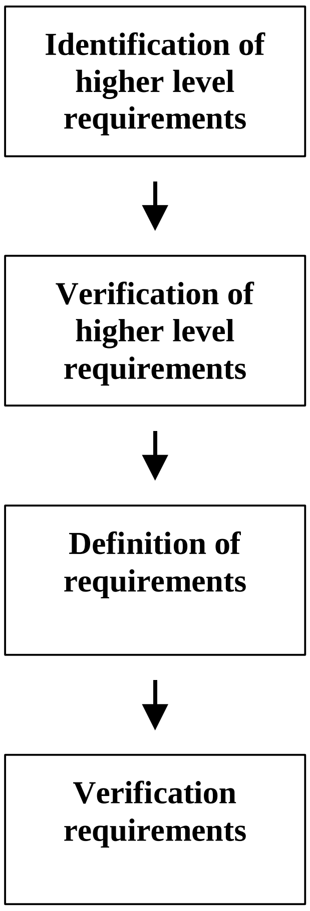
One of the main aspects of avionics systems in general is that they are considered as safety critical. Obviously if the display indicates wrong information like altitude or speed the aircraft may crash. As such two aspects with respect to requirements engineering require special attention:

- requirements related to safety
- traceability of requirements

The requirements engineering process for our division is a process that is primarily focussed on the requirements for avionics community and on transparency to certification authorities. Investigation and experience makes this also a process in evolution. The process is written down into a software requirements standard.

2.1.2 Requirements Engineering Process

The purpose of the requirements engineering process is to come to a complete set of software requirements. The process for the identification of requirements is the following:


Identification of higher level requirements
Verification of higher level requirements
Definition of requirements
Verification requirements

1) Identification of higher level requirements, applicable for the software requirements that need to be defined. This means:

- System requirements allocated to software for the definition of the high level requirements.
- High level requirements for the definition low level requirements

2) The higher level requirements are checked for:

- Consistency: no inconsistencies between the requirements
- For compatibility with the target environment: check if the requirements can be realized on the target system
- Completeness: check whether the set of requirements is complete
- Accuracy

3) Software requirements are then derived from the higher level requirements. This involves:

- Refinement of higher level requirements
- Direct translation
- Reformulation to remove ambiguous formulation
- Completion if this has not been done in system requirements
- Redefinition of higher level requirements into one or more requirements

- Definition of additional requirements

4) The requirements should be checked for and comply with the following characteristics:

- Unambiguous
- Verifiable
- Complete and explicit
- Traceable
- Unique
- Single

Remark: the generation of the low level requirements should actually be seen as part of the software design process.

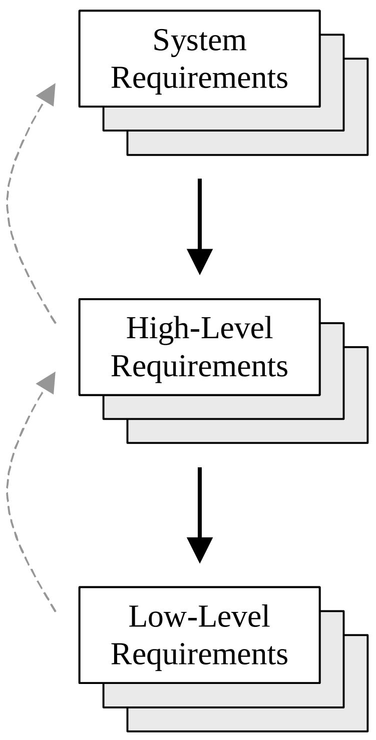
The detailed process guidelines have been inserted in the software requirements standard.

2.1.3 Classification of requirements

In the past there was not a unified way to defined and classify requirements. In some cases this was done based on the chapter/paragraph in the software requirements document, based on a requirement number, based on the product name.

The classification proposed is based on previous experiences and that have turned out to be important, especially in our application domain.

2.1.3.1 Hierarchical classification of requirements



Three main levels can be distinguished in the classification:

- System requirements
- High level requirements
- Low level requirements

Within each level there may be one or more sub-levels or other classifications.

2.1.3.1.1 System requirements

These are the requirement defined at system level. System level can for example be:

- avionics system level: general cockpit display specification defined by the system integrator
- display system level: development specification for the avionics display

It is a general set of requirements not specifically associated to software only. It can include hardware, software, mechanics, quality, …

- Example of a system level requirement:

The primary flight display (PFD) shall display ‘g’ data

2.1.3.1.2 High-level software requirements

High level software requirements are software requirements developed from analysis of system requirements, system architecture, safety related requirements and additional software requirements not coming directly from system level.

High level software requirements should address all system level requirements, at least those requirements allocated to software. This is enforced and verified through the system/high level requirement traceability.

- Example of a high level requirement:

The software **shall** display the numerical G Reading in the range 0.0 to 12.0.

> The PFD shall display ’g’ data in the range of 0.0 to 12.0

2.1.3.1.3 Low-level software requirements

Low level requirements are requirements from which it should be possible to generate the source code in an unambiguous way. They are the result of the refinement of the high-level requirements into separate functions. These can then later on be grouped into a more logical software structure. As such the definition of the low level requirements is to be seen as part of the design process.

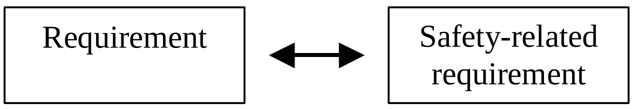
Low level software requirements are software requirements that are derived from high-level requirements, design constraints and additional requirements, not coming directly from high level requirements.

Low level software requirements should address all high level requirements. This is also enforced and verified through the high/low level requirement traceability.

- Example of a low level requirement:

> The resolution of the ’g’ data shall be 0.1

2.1.3.2 Classification based on safety-related requirements



Safety related requirements specify the desired immunity from, and the responses to, failure conditions. These requirements are defined to preclude or limit the effects of faults, and may provide fault detection and fault tolerance.

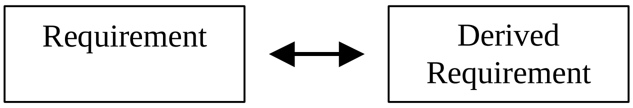
Since safety-related requirements are defined to have an important influence on the safety of the airplane, they need to be treated with more care during development. Especially with respect of verification, more effort is required for example (more) robustness testing.

- Example of a safety related requirement:

> When the 'g' data is invalid, the reading shall be removed and replaced by a red failure indication

Requirement at every hierarchical level can be identified as safety-related. However lower level requirements derived from safety-related requirements become of course also safety-related

2.1.3.3 Classification based on traceability

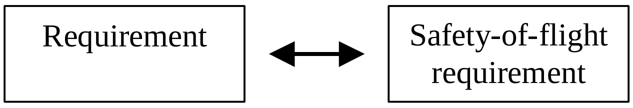


A classification is made based traceability of requirements to higher level requirements. These are the additional requirements resulting from the software development processes, which may not be directly traceable to higher level requirements. Non traceable requirements are called derived requirements.

High and low level requirements can be identified as derived.

The identification of derived requirements is important at system level, since these additional requirements can have an important impact on the system safety and so need to be addressed by the system integrator.

2.1.3.4 Classification based on safety-of-flight requirements



The classification deals with the identification of flight operational requirements. Safety-of-flight are the requirements required for first flight, which is usually the flight testing phase of the aircraft.

Safety-of-flight requirements can be defined at system level or for the high level software requirements.

The identification of safety-of-flight requirements is important in reducing the initial verification effort. The software product can be developed with limited testing for first flight/integration, and thus reducing the regression effort when problems are revealed.

2.1.3.5 Classification based on functional class

The classification based on functional class groups all functionally related requirements. Example:

All requirement related on the Attitude Director Indicator (ADI) define how the ADI behave for incoming data (roll, pitch and yaw) and in case of erroneous or no data.

Grouping requirements allows a better overview and management of the requirements. When not grouping, practice in the past, requirement conflicts can very easy arise when requirements are changing.

In principle a requirement can be divided into several functional classes.

Example:

The requirement describing the behaviour of the ADI in case of wrong input data, is an ADI-related requirement as well as to the error handling capability.

deals with the identification of flight operational requirements. Safety-of-flight are the requirements required for first flight, which is usually the flight testing phase of the aircraft.

2.1.3.6 Classification based on product/product family

2.1.3.6.1 Product requirements

These requirements are driven by the customer, being an external customer, ordering the product, or an internal customer, like the test division responsible for the qualification testing of the product.

Customer for example would like to see a message to be transmitted when a failure occurs on the unit. The test division is more interested in have a discrete output set, triggering some external electronics, when the failure occurs.

2.1.3.6.2 Product family requirements

The product, as part of the product family, 'inherits' the requirements defined on the product family. These are the requirements that form the generic product specification.

- Market survey
- Strategic decisions
- Technology decisions

2.1.3.7 Example of full classification

The following requirement can be defined for the MCDU (Multifunction Control and Display Unit), this is a text based display unit mainly used to enter navigational data.

(P9813-HL-BIT-65-S) In case of a cold start, the MCDU **shall** execute a Power-up Built-In Test.

This requirement is classified as:

- Product name MCDU, which is identified by the specific number (P9813)
- High level requirement, identified by HL
- Safety-related, identified by S
- Contained in functional class BIT

2.1.3.8 Requirements standard

The new classification scheme is currently being integrated in the BarcoView - Avionics software requirements standard. The following is an extract of how this has been integrated in the standard.

1.1.1 Structure and identification of requirements

This paragraph details how requirements shall be structured and identified in the SRD. This assists in managing requirements.

Rule 1: A requirement shall start with a unique identifier tag, followed by a definition part, ending with a paragraph terminator. An optional description and rationale shall each begin with a new paragraph.

Rule 2: Each requirement shall be uniquely identified by a tag that is composed of the following fields, separated by a hyphen, and put between brackets:

(1) the project code
(2) the characters ‘HL’
(3) an optional field to group requirements per mode or per main function. A list, specifying the groups, shall be defined per project, during the requirements process and clearly documented in the SRD.
(4) a unique sequence number, as explained in the rules below
(5) optionally the character ‘S’ (to indicate a safety-related requirement) followed by ‘D’ (to indicate a derived requirement).

Rationale: Examples:

- *(P0216-HL-33)*
- *(P9935-HL-18-S)*
- *(P9916-HL-57-D)*
- *(P0046-HL-42-SD)*
- *(P0046-HL-MNT-42-D)*

Rule 3: The requirements shall be numbered sequentially, starting from 1, without gaps in the numbering sequence

Rule 4: When a requirement becomes obsolete, its definition, description and rationale shall be replaced by the word ‘Deleted’. The number (and tag) shall not be recycled.

An example of a tag of an obsolete requirement:

- *(P9853–HL-42) Deleted*

2.1.4 Requirements Engineering Tools

2.1.4.1 Representation of requirements

A requirement is identified with an unique identifier tag, followed by a definition part, ending with a paragraph terminator. An optional description and rationale shall each begin with a new paragraph

For the definition part of the requirement, structured natural language is used as the default method. The required behavior may be expressed as a sequence of inputs and outputs, using examples where needed.

(P9517-HL-35)	*When the power was switched off for less than 5 seconds, the CDMS **shall** perform a warm start.*

2.1.4.2 Documentation of requirements

2.1.4.2.1 Software requirements document

In the current approach, the main source for requirements is a software requirements document. The document captures the requirement definition and the traceability to higher level requirements.

The documentation guarantees:

- Definition with the identification scheme as specified in the requirements standard
- Configuration management (CM), through the CM on documents. Each document has its own version.
- Change control
- Traceability

A more in depth description is given in D3.2.1.

2.1.4.2.2 Traceability and review database

In an integrated approach, with requirements on several levels, traceability of requirements, review activity on requirement definition and testing activities, maintaining information in separate documents becomes quite complex. If we also take into account requirements over product families and (software) components, it becomes a serious exercise to keep everything in sync and manageable.

In order to cope with the integrated approach, the Avionics division has started with the development of a requirements traceability database. The purpose of the database is to capture development artefacts like:

- System requirements
- Software requirements
- Design artefacts
- Review artefacts
- Test artefacts

A more in depth description is given in D3.2.1.

2.1.5 Status/Open Issues

2.1.5.1 Status

The software requirements standard has been defined and starts to be implemented in some projects. With the results of the practical implementation and the experiences of other partners in the Empress consortium, Barco wants to:

- Improve its requirements engineering process
- Refine its classification scheme

2.1.6 Open issues

In the proposed approach there is CM is foreseen on document level but CM on individual requirement level is not foreseen. We can only able to track changes to requirements by tracking the changes in the different versions of the documents. At the end this can become quite cumbersome and error prone, especially when the stability of the requirements is very low.

This is recognized as an important shortcoming and is considered as a next step forward.

2.2 Classification of Requirements in the Home Domain

2.2.1 Introduction

Jabil Circuit Hasselt is specialized in product design and manufacturing of communication, mulitmedia and other (emerging) digital products. Key products at this moment are digital set-top boxes, DVD and DVD+RW players, LCD TV.

Typical Jabil experiences in the area of embedded software development are:

- The embedded systems are not longer purely stand-alone systems; they often need to be connected in a network or have to interact with all kinds of devices. Addition of new devices to the environment will also have an impact on the embedded software. Systems should be built flexible enough to adapt to new elements in the environment.
- Home domain products are typically organized in product ranges and product families. These products differ with respect to featuring or product design. They are often positioned in the market at different price points or in different geographical areas. Each specific environment may have different requirements and may need a slightly different version of the embedded software.
- Besides the growing complexity, the amount of embedded software and thus the amount of requirements is growing every year - a substantial part of this growth stems from more elaborate featuring and the addition of software intensive functionalities.

To summarize, in general Jabil Hasselt has to control large amounts of continuously evolving requirements for products, having complex relations to the environment and being part of product families. For Jabil it is essential to find ways to classify these requirements.

The investigation of Jabil Hasselt resulted in classifications on two levels:

- The Jabil RE process classifies requirements according to their abstraction level
- For product families, requirements are classified according to product family related information.

The classification according to the RE process is supported by a documentation structure: for each type of requirements a corresponding document type is defined. This is described in section 2.2.2.

The classification of requirements according to product family information is specific for each product (family). An appropriate structure has to be defined at the beginning of each product family development project: e.g. specific types of requirements or attribute types. This classification provides the basis for the relation between classes of requirements in product families and architectural decisions. This is described in section 2.2.3

2.2.2 Classification according to the Jabil RE process

The requirements engineering process in Jabil contains three sub-processes: elicitation, analysis and specification. The process is described as a generic process meaning that the process description can be applied to several abstraction levels of requirements. These levels of requirements are the first classification of requirements in Jabil and a base for traceability.

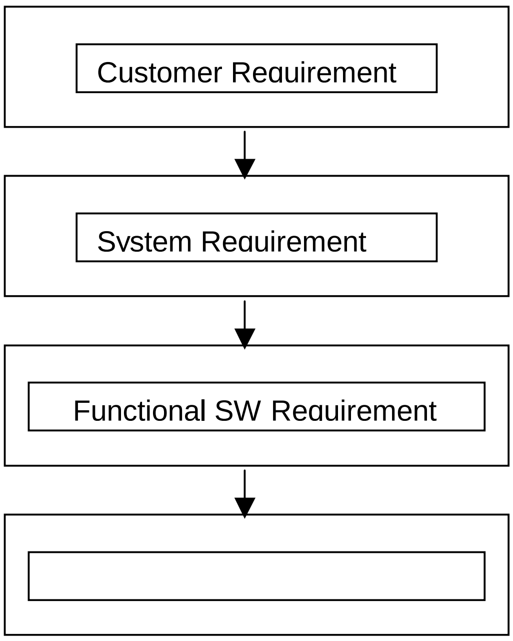


Figure 4: classification according to the RE process

- Customer requirements (CR): these requirements are coming from the customer. The customer can be an external customer or an internal customer. An internal customer is e.g. a product manager. In general customer requirements are high level requirements, poorly specified.

- System requirements (SR): System requirements are derived from the customer requirements. In parallel to the system requirement specification, a top level design or architecture is developed. From customer requirements and architectural information engineers derive customer requirements for the complete system. No aspect of the system may be forgotten. The system specification functions as a contract to the customer.
- Functional software requirements (FR): the system requirements assigned to the software are specified in more detail. The architecture and the system specification are used as input.
- Module requirements (MR): detailed requirements for the software modules, derived from the functional requirements and the architectural information.

Each requirement is classified according to this high level classification. Customer requirements are described in a document called “Customer Requirements Specification”, system requirements in a “System Requirements Specification”, functional software requirements in a “Functional SW Requirements Specification” and module requirements in a “Module requirements Specification”. As mentioned, this classification is a basis for a traceability structure.

As the names of these documents suggest, the documents should contain requirements. However, in practice it not always clear whether a statement containing some design information belongs in a requirements document or not. This is the reason why Jabil needed a kind of reasoning (pre-classification ?) to decide on this problem, as described here:

Requirements describe what a system or subsystem should do, and not how it should do it. Design decisions or design solutions describe how requirements can be realized. Requirements, the “what”, belong in a requirements specification. Design solutions, the “how”, belong in a design description, e.g. a Top Level Design (TLD) or a Detailed Design (DD).

“What” and “how” should not be mixed, because early design decisions may prevent the requirements engineers from finding the real purposes, the requirements of the system. Early design decisions also restrict the solution space for the architect/designer, and may prevent him from finding a good solution.

This is the theoretical background, important to keep in mind.

But our practice is more complex. It happens that a design solution is explicitly asked by the customer to be part of the system. In that case, this design solution is a requirement; it belongs in the requirements specification.

Examples:

(1) “The interface between CPE (Customer Premise Equipment) and Host PC shall be Ethernet 10/100m, wired.”

(2) “The CPE (Customer Premise Equipment) shall be operated by the LINUX OS.

To decide whether a design decision belongs in a requirements specification or in a design description it is important to know the source of the decision. When a design solution comes from the customer, and the customer is aware of the restrictions of it, it is a real requirement. But the requirements engineer, the author of requirements specification must always be aware not to take premature design decisions and to write them as a requirement in a requirements specification. Required design decisions belong to the non-functional requirements.

Besides these design decisions, our requirements documents can also contain another group of non-functional requirements, the so-called “ility” statements. These are statements on reliability, usability, efficiency, maintainability or portability. “Ility” requirements put also constraints on the design or implementation of the system or subsystem.

Examples:

(1) "The execution time of the Diagnostic Software tests for the quality department shall be at most 3 seconds." (efficiency)

(2) "When a replacement CPE (Customer Premise Equipment) is available, the CPE can be replaced and (re-) configured within 2 hours." (maintainability)

2.2.3 Classification of product family requirements

The requirements for products belonging to a product family are classified according to a structure specified at the beginning of the development project.

In this case, however, we do in fact have several product lines (or several families of products) all based on the same platform. For example, one product family is the TV products, and the Monitor products are another family of products. Each family of products have three (main) types of requirements:

1. Basic requirements – These are requirements that will not change and will be in (almost) all products. For example, for a TV the tuner is basic while it isn't for a monitor. However, the VGA connector is basic for a monitor but optional for a TV. In this is called commonalities.
2. Optional requirements – Depending on the specific customer, several options can be added to the products. Optional here means requirements that somehow add extra functionality besides the basic features required to have a operational product. This is one part of the variabilities. This can be for a TV a second tuner or a remote control for a monitor.
3. Variability requirements – Almost all requirements can be implemented a little different (e.g. depending on region). For example, in Europe PAL is mainly used, while in the US mainly NTSC. Hence, for the basic requirement for a TV that it should have a tuner, there are variations in the concrete type tuner needed. The types of variations that are possible are the following:
    - Single value – of the sub-requirements, only one can be chosen. For example, only one TV/RF standard can be supported by one tuner.
    - Multiple values – sometimes it is possible to choose multiple options from one requirement, e.g. there can be more than one connector.
    - Numerical – some variability can be in the form of a number, e.g. number of buttons on TV.

The main idea is that basic requirements should not change for different products within one family (e.g. for a TV). The variations are realized either by the optional requirements or the variability requirements, e.g. exclude a second tuner and use PAL BG. The different types are necessary to understand how the tool support should be implemented and what kind of information should be in the requirements database.

2.3 Abstraction Layers for Specifying Evolving Software Systems in the Automotive Domain

This chapter introduces an abstraction layer model for the specification of requirements of evolving software systems and deals with handling of evolution of requirements. For these tasks, different methods are described. The mapping of these methods to the identified acitivies and sub-activies of the EMPRESS-Process discipline 'Requirements' as presented in chapter 1 of this deliverable is shown in the following table (Table 2-1).

RE Activity	RE Sub-Activity	RE Method covered by chapter 2.3
Elicitation	Define Scope	N/a
	Collection	Identification of further reasons upon the basing on use cases (communication between developers and stakeholder)
	Common Understanding	Common understanding by use cases and feature list
	Negotiation	N/a
Analysis	Classification	Abstraction layers
	Verification and Validation	N/a
	Identification of discrepancies	N/a
Documentation	Visualisation of contents	Text, Use Cases, Feature List
	Structuring of contents	Structuring concerning the abstraction levels.

Table 2-1: Mapping of chapter 2.3 methods to the RE activites substructure

The three touched activities elicitation, analysis and documentation are principal elements of the sub-discipline 'Requirements Engineering' (RE) of the EMPRESS-Process discipline 'Requirements'.

2.3.1 Context of Subsystems in Automotive Industry and Evolution

The automotive industry obviously deals with evolving systems. To make the challenges we face today clear, Figure 2-5 depicts what we call "product dimensions". In one dimension, we have different products like the Mercedes-Benz E-class or S-class, having different electronic control units (ECUs) with different features. A second dimension is that of product variations, i.e. for one product (e.g. E-class), there might be differences specified for different markets (e.g. EU, USA, Japan), for different customer equipment (e.g. entry-version, mid-version, high-version) or even other variants (e.g. Limousine, Cabrio, Combi, …). The third dimension is that of time, i.e. different releases of products in time.

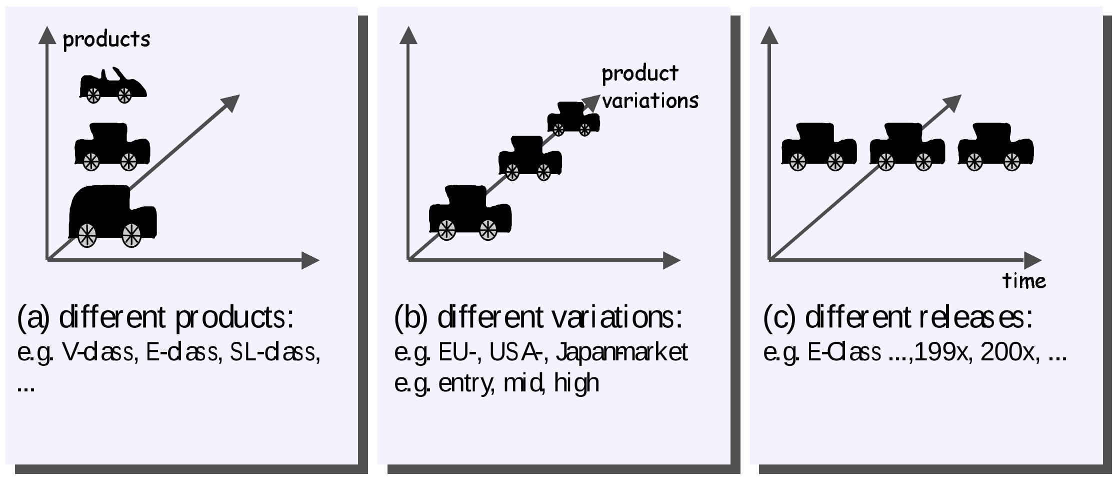

products
(a) different products:
e.g. V-class, E-class, SL-class, ...
product variations
(b) different variations:
e.g. EU-, USA-, Japan-market
e.g. entry, mid, high
time
(c) different releases:
e.g. E-Class ...,199x, 200x, ...

Figure 2-5: Typical product dimensions in automotive industry

For a domain expert responsible for specifying a single subsystem (e.g. the wash/wipe-ECU) this means that he has to provide multiple specifications (e.g. one for the E-class for the US-market in 199x, another for S-class for EU-market in 200x). From the view of one subsystem the picture might look something like Figure 2-6.

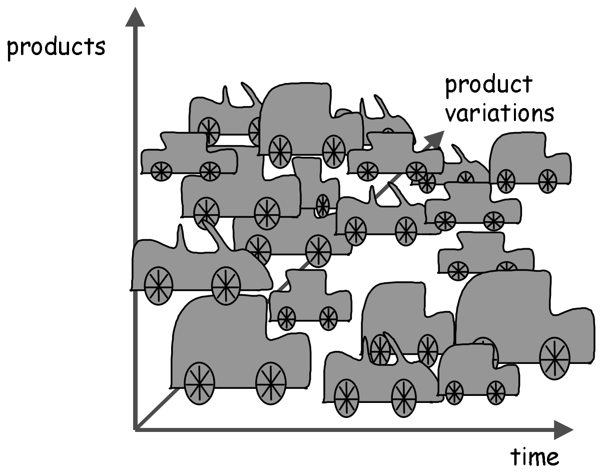


Figure 2-6: The view of a single subsystem

For specifying a new component (ECU) in this situation (the situation of having already specified that ECU many times before in the product/variations/time-cube), it's possible to reuse the requirements in some way (compare D1.1, 2.1.1.1 Elicitation). Reusing such existing requirements can be done in more or less advanced ways. These are classified in Figure 2-7 according to the degree of how systematic they are.

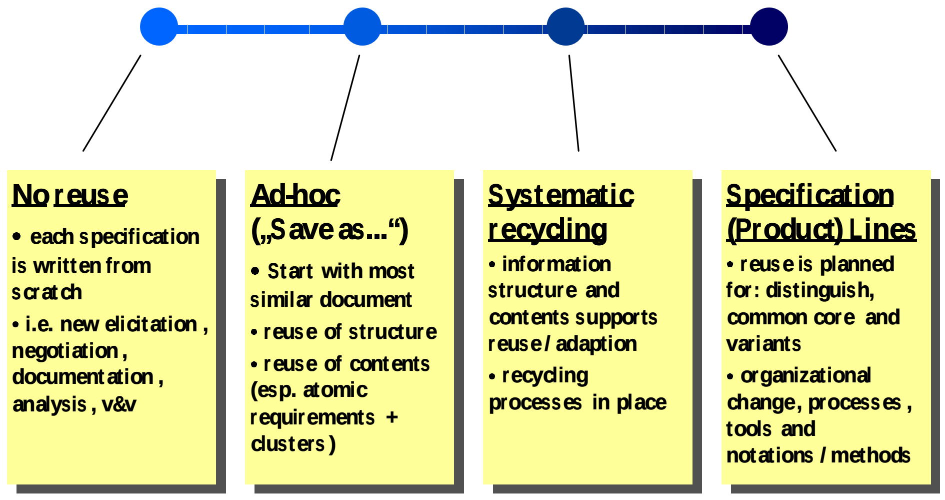

No reuse
• each specification is written from scratch
• i.e. new elicitation, negotiation, documentation, analysis, v&v
Ad-hoc („Save as…“)
• Start with most similar document
• reuse of structure
• reuse of contents (esp. atomic requirements + clusters)
Systematic recycling
• information structure and contents supports reuse/adaption
• recycling processes in place
Specification (Product) Lines
• reuse is planned for: distinguish, common core and variants
• organizational change, processes, tools and notations / methods

Figure 2-7: Reuse scale for specifications

Our engineers today most often use the “ad-hoc” method: They start by using the specification document most similar to that to be written. Then they delete the requirements which aren’t valid any more, add others from other specifications or new ones, and modify existing requirements. This method is much more intelligent than not using the existing specifications (“no reuse”). On the other hand, this method is today not very systematic because a specification document today exists of many requirements organised in a flat way (i.e. no hierarchy of abstractions), which could give fast and reliable understanding of information clusters in the specification. This is the core idea and demand for stage 3 in the reuse scale: providing suitable abstraction layers in the specification to support a systematic approach in recycling requirements for evolving systems. The abstractions layers are therefore defined and explored in the following chapters.

The most sophisticated way in reusing requirements for a product family for sure is that of planning product variants and defining a common core for requirements, i.e. establishing a specification (product) line. This approach is described in another chapter of this document. Our hypothesis is that the product line approach, too, will be most beneficial when using the framework of abstraction layers for specifications.

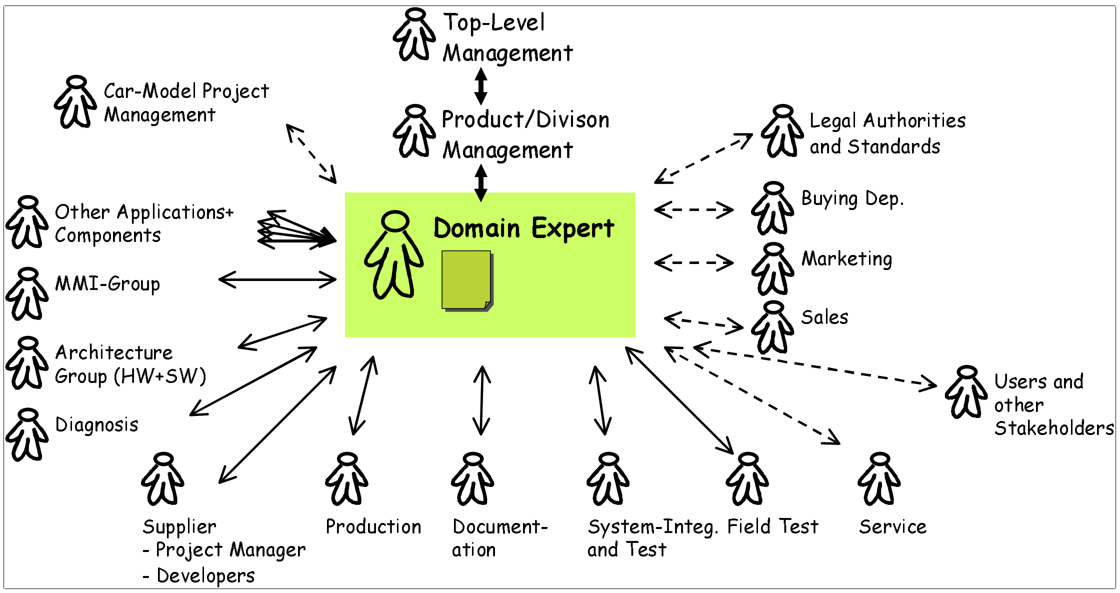


Figure 2-8: Different information for different stakeholder: Direct and indirect communication

of a domain expert
responsible for a specification document (the figure is fictitious and shows not an actual organisation)

So far we argued that abstraction layers are needed to cope with evolving systems in a complex product context. To make the picture more complete, it is also notable that abstraction layers should also be seen in terms of information demands of different stakeholders. A typical (yet fictitious) example is given in Figure 2-8 which shows possible stakeholders in a subsystem specification document, i.e. in a domain expert’s work product. Seen from this point of view there’s again an argument for abstraction layers as e.g. product Management needs different information than field test – and in any way every stakeholder must be able to understand and navigate through the specification within a reasonable time span. This stakeholder views might also shape the demands upon the abstraction layers, i.e. the definition of which abstraction layers shall exist, which information they shall contain and which relationships shall be defined and maintained.

2.3.2 Abstraction Layers

2.3.2.1 Definition of the Abstraction Layers

Our framework for requirements defines three abstraction layers (or levels) of requirements, where each layer corresponds to a basic stakeholder-view on the system, namely management, user, and developer. Although (as Figure 2-8 implies) there are many different stakeholders of the system each having a slightly different viewpoint, we don’t assume that each group of stakeholder needs its specific abstraction layer (as e.g. viewpoint analysis does in some way [SSV98]), but structuring the specification by establishing three main views on the system helps all groups. Additional stakeholders may get specific and appropriate views by extracting information from several of the main layers. The three levels are (a similar definition can be found in [Wie99]):

- Business requirements (BR)
- User requirements (UR)
- System requirements (SR)

Business requirements (BR) name the essential strategic goals of the system (or sub-system), its scope and general constraints. This layer contains the information relevant to the customer’s product management and/or marketing department, i.e. the requirements that must be met in order to stay in the market (or help the overall product to stay in the market).

The rationale for explicitly stating the BR is:

- Details can‘t be understood without the (top) goals.
- Without a clear, understandable and agreed focus a project will run out-of-control.
- For a goal many solutions exist (in this case: possible sets of user requirements)—if the goal isn‘t stated explicitly and agreed upon, there will be endless discussions and wrong decisions.
- For most of the automotive components the business requirements are highly stable and therefore systematic reuse on this level is very easy. If a BR is changed, it is obvious which URs (and which SRs) are potentially affected by analysing the RATIONALE relation (see below).

User requirements (UR) name what the user (meaning a human role or other systems) essentially desires from the system. The system is viewed as a black-box, such that this layer only describes system functions the system provides and the user problem which is solved through the system. This layer thus contains what is relevant to a user of the system.

The rationale for extracting UR is:

- If the user problem/goals are mixed with technical details it's a confusion of goals and means to achieve the goals. The document is then not clearly understandable anymore.
- For most of the automotive components the user requirements are highly stable and therefore systematic reuse on this level is very easy. If a UR is changed, it is obvious which SRs are potentially affected by analysing the RATIONALE relation (see below).
- Without this layer the set of system requirements (see below) wouldn't be manageable (release planning, test planning, status tracking) and if the system evolves it would not be reusable in a structured way.
- A phased RE process is supported, i.e. a process which first defines and reviews goals before specifying means to achieve goals.

System requirements (SR) specify the system, i.e. they define **how** the user requirements are to be met by a specific system solution in such a form that it can be handed over for development. Thus this layer describes what is relevant to the developing organisation (which could be a supplier).

The rationale for writing down SR is:

- The system requirements build the basis for the implementation of the system.
- Without this kind of requirements it would be nearly impossible to test the system on its make or brake.

The layers together form a pyramid (Figure 2-9), which means that:

- Business requirements lead the way to the user requirements which again lead the way to system requirements. Or said the other way around: system requirements are to fulfil user requirements which are to fulfil business requirements.
- The amount of information in the BR-layer is much less than in UR-layer, and in UR-layer much less than in SR-layer (an example ratio could be 1 : 10 : 100 pages for the BR:UR:SR). This also should correlate to the time needed to understand the information, i.e. understanding of the BR should take 10 times less time than understanding the UR.

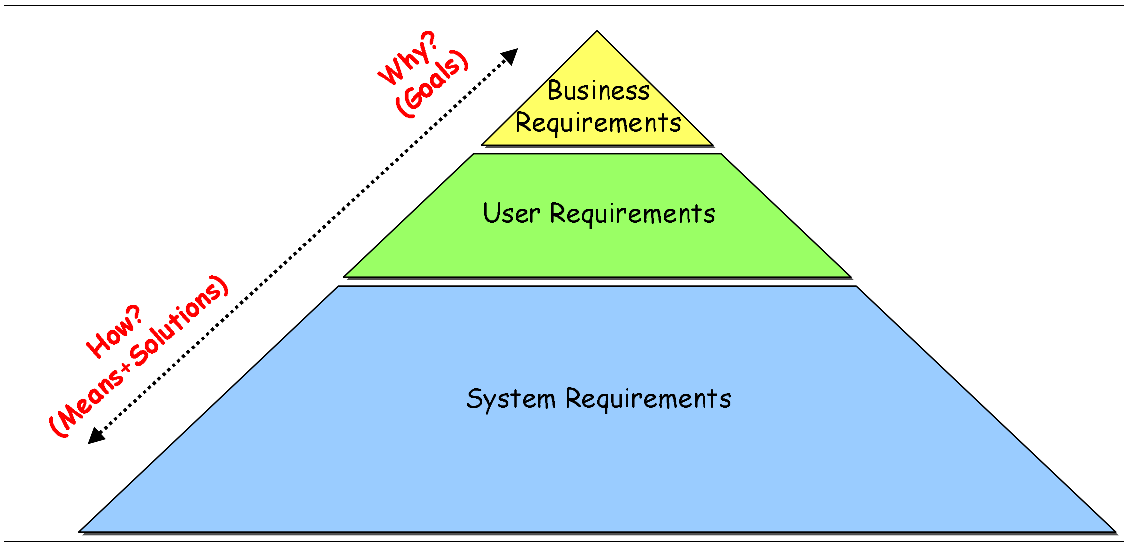


Figure 2-9: The abstraction layers pyramid

2.3.2.2 A Specification Concept for the Abstraction Layers

The 3-layer-model is an abstract meta-model to structure specification contents. For the context of evolving systems, we now define a more concrete concept of how to specify the contents of the layers. The basic idea is depicted in Figure 2-10.

The business requirements are documented in a Vision&Scope-module (module meaning an information cluster, which might be a document or part of a document). The parts of this module are:

Part	***Description***
"Vision"-Paragraph	Top-Goal(s) What we want to reach Why we want the product or sub-system
"Scope"-Paragraph	Scope Solution-Space On what kind of solutions we will work on
"General Constraints" (optional)	Are there design solutions which must be used or are not acceptable? (regarding the product but also the development process; e.g. quality, time, cost)?
"Stakeholder List" (optional)	Who has an interest in the product or is affected by the product?

Table 2-2: Parts of the Vision&Scope module

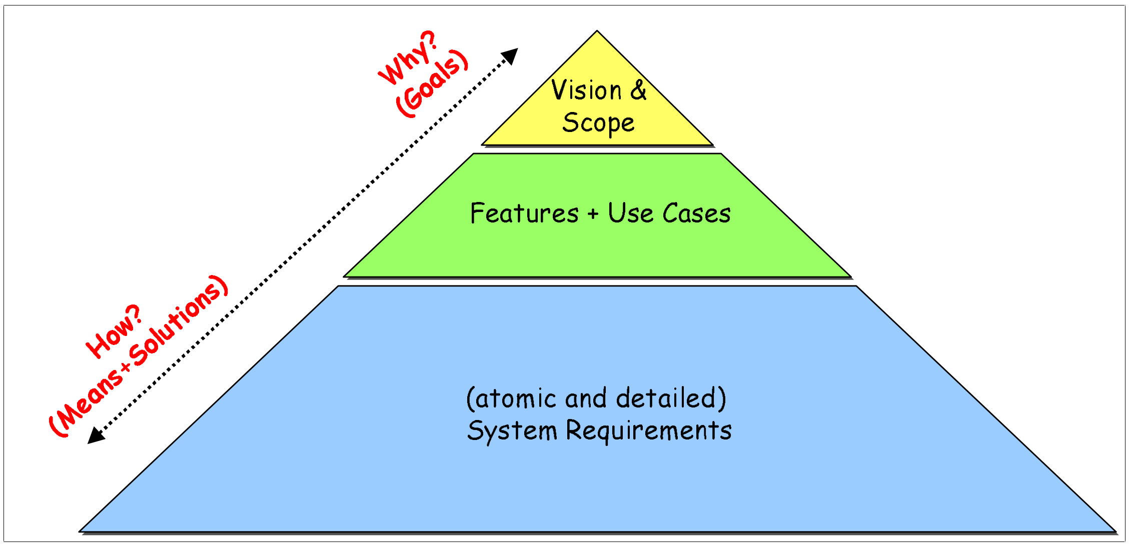


Figure 2-10: Specification concept for the abstraction layers pyramid

The user requirements layer might be modelled in our concept in two forms: features and Use Cases.

Features are system functionality or system design properties desired by the stakeholders. Features can be conveniently organised as a hierarchical list. They especially support product management with planning and managing system parts. Features are by definition static properties and are most often already designed solutions.

Use Cases come in many different forms and styles (see e.g. [AZ02]). In this context we view Use Cases as descriptions of the user (or human role or a system) problem domain. In the Use Cases a user uses system features in a chronological manner to accomplish his goals.

As [CDK01] already showed features and Use Cases can complement each other and there might be different strategies for using them, i.e. there might be a feature-driven project or a Use Case-driven project.

2.3.2.3 A Possible Information Model

Figure 2-11 shows an instantiated information model for the specification concept. The model defines (similar to an entity relationship-model) information entities (here named as “modules”) and relationships between the entities. Different information models are possible depending on project demands. In the information model we’ve chosen features as the central part of the user requirements and Use Cases are “just” there to describe and clarify the user interactions/problem domain of the system. Thus it can be thought of as a “feature-driven”-approach (cf. [CDK01]).

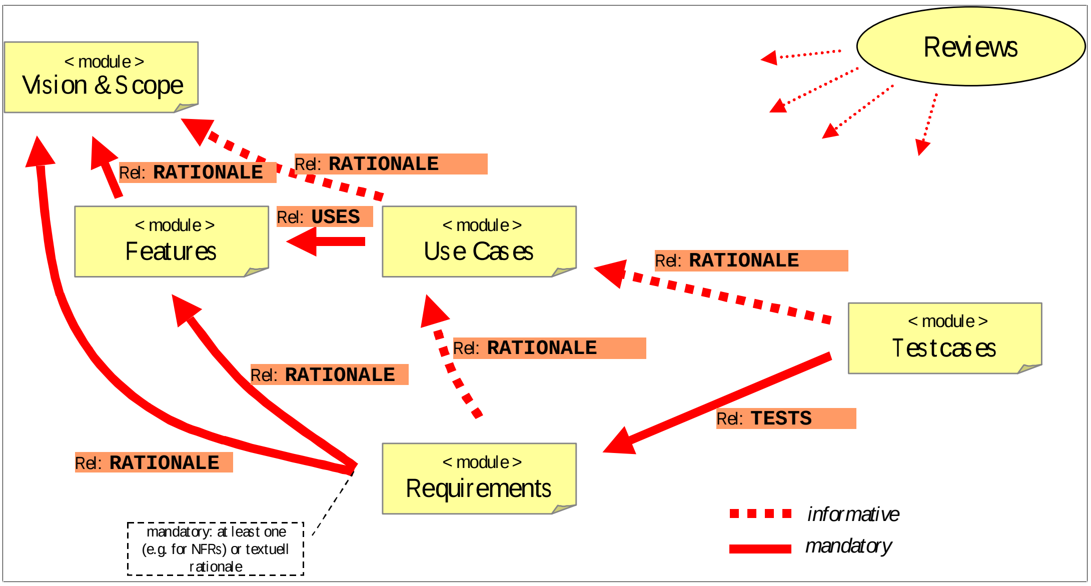


Figure 2-11: A possible information model for the abstraction layers (feature driven)

The following table describes shortly the single relations between the information entities (it’s worth to note that in principle each relationship is a many-to-many relationship):

Source	***Destination***	***Relationship***	***Description***
Features	Vision&Scope	RATIONALE	A feature is traced back to a goal stated in vision&scope, i.e. a justification is that it is part of achieving a business requirement.
Use Cases	Vision & Scope	RATIONALE (informative)	A Use Case is traced back to a goal stated in vision&scope, i.e. the existence of this specific Use Case is justified by a business requirement. This relationship is „informative“ because not every Use Case must have a business

			requirement as a rationale.
Use Cases	Features	USES	In a Use Case step a feature is used by an actor (human role or other system) to achieve a user goal.
Requirements	Vision&Scope	RATIONALE	A requirements justification is probably directly derived from a business requirement. This is especially true for global non-functional requirements.
Requirements	Features	RATIONALE	To a feature a set of detailed requirements must be derived to specify the feature, so that the feature justifies the requirements.
Requirements	Use Cases	RATIONALE	Detailed requirements might be better understood when a Use Case is given that justifies that requirement. This relationship is „informative“ because not every requirement must be justified by a Use Case.
Test Cases	Use Cases	RATIONALE	Use Cases are used to elicit test cases, so test cases might be better understood by stating the Use Cases they are derived from. This relationship is „informative“ because not every test case must be derived from a Use Case.
Test Cases	Requirements	TESTS	Each requirement must be tested by at least one test case.

Table 2-3: Relationships of the Information Model

The goal of the relationships is to support the requirements engineering process, i.e. all activities directly or indirectly related to the specification of a (evolving) subsystem. As Figure 2-8 has shown, these activities are manifold, i.e. the communication needs are manifold. But the relationships make it possible to firstly reason about the justification of existing requirements and secondly to reason about the impact of requirements. Both are crucial for coping with evolving systems. The relationships can be made visible in a requirements management tool through the definition of “views”, which show entities related through a relationship to another entity. These views thus support the workflow of the requirements engineering process. Possible views are depicted in Figure 2-12.

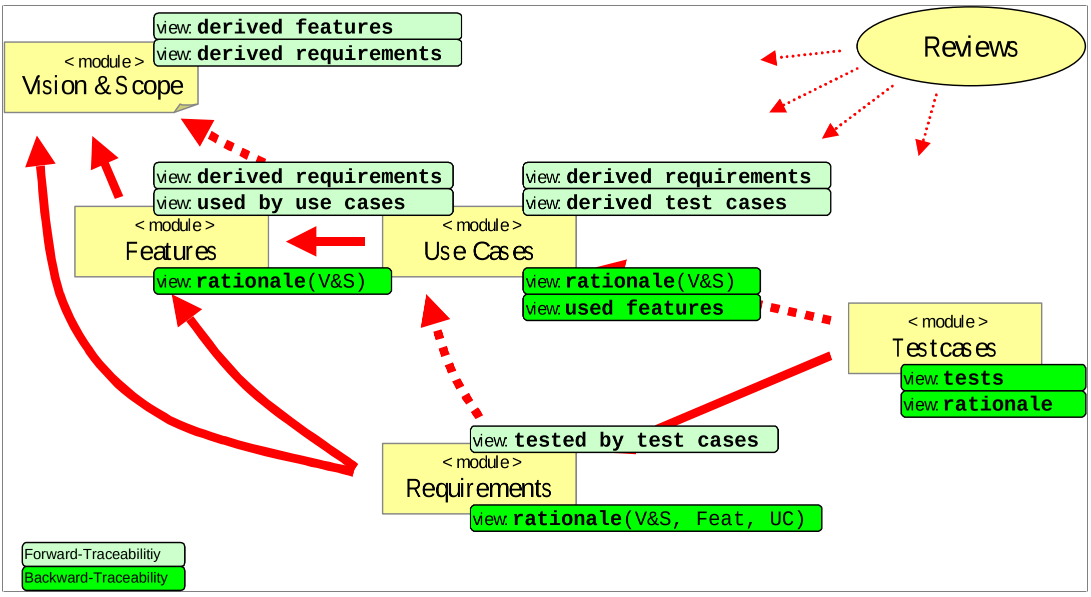


Figure 2-12: Views supporting the RE process for evolving systems with forward and backward traceability based on the relations defined

2.3.2.4 Rules Framework for the Specification Concept

Just like it's wise to first state requirements before building a product, it's also wise to state the requirements for the specification concept, i.e. to give a guideline which information is expected for the different information entities and how they should be stated. The rules can be thought of as a "policy" for requirements engineering. This idea is not new: Tom Gilb sees such rules as part of a standards set for requirements engineering [Gil02]. Figure 2-13 gives an overview of the framework for the defined specification concept.

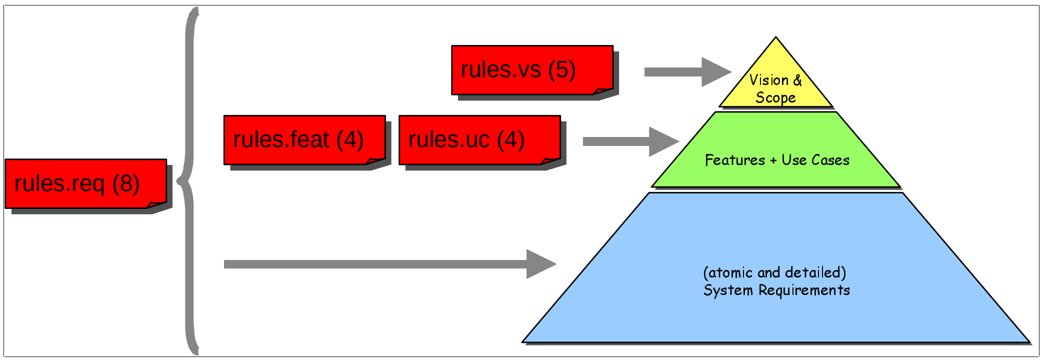


Figure 2-13: Rules framework for the Specification Concept (the numbers are the amount of rules)

2.3.2.4.1 Generic Rules (rules.req)

The following rules are generic, i.e. they are applicable to all requirements documents. In particular these rules are partly based on the work of Tom Gilb [Gil02].

Consistency (rules.req1)
Each requirement is consistent with all other contents in the document and is consistent with

all related documents (e.g. sources, references, derived requirements and implementations, ...).

Rationale: Inconsistencies can lead directly to wrong decisions and implementations, i.e. a “major defect”.

Bad example: “Radiosystem”:
- a Feature is defined as “Station memory consisting of 30 stations”
- in the accompanying requirements “User selects one of the 15 stations”

Clarity (rules.req2)

The specification entities are clear and unambiguous for all intended stakeholders, i.e. no misunderstandings are possible and the requirements are testable.

Rationale: Unclear contents leads to wrong decisions and defects in the development, if they are not clear to all stakeholders (e.g. supplier, test, documentation, management, ...).

Bad example: „All functions should be adaptable by the means of parameters to fulfil different demands.“
→ which functions? should?? adaptable to what extent? how are parameters given? fulfil which demands?

Goals vs. Means (rules.req3)

Goals (or problems to be solved) are clearly distinguished from means (solutions, design solutions) to accomplish these goals. The specification contains only necessary means, i.e. justified design constraints.

Rationale: If goals and means are mixed, they are no more distinguishable which leads firstly to wrong decisions or sub-optimisations and secondly to obscurity (see rules.req2)

Example: „Wash/wipe functionality of the windscreen wiper“
Goal: “Clear sight for different rain intensities”
Means-1: “2 wiping speeds and intermitted wiping”
Means-2: “Automatic wiping speed (rain sensor)”

Rationale (rules.req4)

Each requirement is justified through a superior business goal, a user need, a design constraint or a reference. The rationale must be obvious.

Rationale: A not justified requirement might be not anymore valid (especially when reusing requirements for evolving system), thus it might lead to unnecessary consumption of cost and time when further using the specification downstream in development, documentation etc.

Example: Valid rationales in this sense are:
- A requirement is stated in a feature section and it’s obvious that the requirement details the feature.
- A requirement is linked in a tool to a Use Case, whose steps show in which situation the requirement is necessary.
- “Implied by Standard xyz , chapt. 3.2.1 (see reference)”
- "Decided by Marketing Management, dep. ABC, dd.mm.yyyy”

Maturity and History (rules.req5)
The maturity (status) of requirements (resp. sets of requirements) is always obvious and the history of the requirements can be reconstructed any time.

Rationale: If immature content is used as a basis for further development, this it might result in wrong decisions, in defects or in needed rework. If the history can't be reconstructed, the same discussions might come up again many times downstream.

Example: A valid approach would be to give each requirement a status attribute, which values "initial, specified, agreed, rejected" and to let a tool keep track of the change-history of objects.

Referenceableness (rules.req6)
The specification contents are divided into atomic elements, which can be referenced individually and stable over lifetime of the entity. The elements are themselves logical and meaningful.

Rationale: Without a possibility of referencing specific elements of a specification, effective and efficient communication about these elements is not possible (e.g. in a review, e.g. in customer-supplier-relationship). Furthermore it's not possible to make statements about the elements, i.e. to give the element a state, an author, a rationale, a review comment, a test case which tests it etc.

Example: A valid approach is to use a tool for creating and managing IDs for single objects (still the requirements engineer is responsible for a reasonable splitting of paragraphs into single objects).

Requirements vs. Comments (rules.req7)
All commentary information not part of the actual specification contents (e.g. comments, todos, tbds, ideas, etc.) are separated from the requirements through appropriate means.

Rationale: Otherwise comments might be taken for requirements by somebody else, which might lead to wrong decisions and defects. Mixing the requirements also leads to obscureness (rules.req2).

Example: A valid approach is to use an attribute in a requirements management tool.

Minimal Redundancy (rules.req8)
All specification contents are (as far as possible) only specified in one place and reused everywhere else via references. This applies to single entities as well as whole specification documents.

Rationale: Redundancy leads to inconsistencies (see rules.req1) and thus to defects.

2.3.2.4.2 Rules for Vision&Scope (rules.vs)

Short (rules.vs1)
Vision & Scope is short.

Rationale: Everybody must be able to read and understand V&S in less than 10min.

Success Criteria: One DIN-A4 page (for a common subsystem).

Understandable (rules.vs2)
Vision&Scope is easily understandable for every stakeholder of the product or component without additional comments or documents.

Rationale: Otherwise the risk is too high that different persons in the project and management have a different understanding of V&S, thus making wrong decisions or having wrong discussions.

Success Criteria: Given to somebody never concerned with the specific project/product before (but with the general domain knowledge), that person understands V&S nevertheless

Customer view (rules.vs3)
Vision&Scope contains the information that matters for the customer/(project-) management, i.e. it contains at least the top-goals (vision) and a definition of the product scope.

Rationale: This view/abstraction guarantees that one can quickly understand the why behind the document (thus, the whole document can be used for decision making, for communication, and for quick and structured recycling of the document)

Success Criteria: One could construct a management slide (for decision) from the Vision&Scope

Optional: This view could also contain General Constraints and a list of the Stakeholders.

Quantified (rules.vs4)
Important qualitative statement (like „better“, „easy to use“, ...) shall be quantified in an appropriate way.

Rationale: General qualitative terms are ambiguous, i.e. they can mean anything, thus different persons will have a different understanding, leading to wrong decisions.

Success Criteria: Either a scale and meter is defined, or some benchmark is given (e.g. target goal, minimum, past) (see [Gil02] for more explanation on quantifying qualities)

Generic (rules.vs5)
The generic rules for specification (rules.req) are applied.

2.3.2.4.3 Rules for Features (rules.feat)

Essential desire (rules.feat1)
Each feature names an essential system functionality or design property desired by a stakeholder which traces back (directly or indirectly) to a business requirement.

Rationale: To keep the feature set manageable (e.g. to negotiate), the set of features has to contain only the ones that matter. Details can be left to „local decision makers” like domain experts.

Success Criteria: A feature is something a product/marketing manager (and most often the user) would say: „I want this in (or out of) this product release“

Comment: The stakeholder is typically the user, but could also be e.g. service, diagnosis or production.

Unambiguous but short (rules.feat2)
Each feature has a meaningful name and a short description. The name and description do include as little technical details as possible.

Rationale: (1) Ambiguous feature names without description will lead to wrong decisions (2) Blurring features with unnecessary technical details will make a quick overview impossible.

Success Criteria: (1) All decision makers involved in the development project (e.g. product manager, marketing dep., and supplier) understand the feature without additional information quickly. (2) All decision makers [...] can quickly get an overview of the whole feature set they are concerned with.

Manageability (rules.feat3)
Each feature has appropriate attributes to plan and manage them. As a minimum, a responsible person and the status of the feature is given.

Rationale: Features are used to manage the development, i.e. to plan, negotiation, and control.

Success Criteria: No additional Excel-sheets or similar have to be used for project management.

Generic (rules.feat4)
The generic rules for specification (rules.req) are applied.

2.3.2.4.4 Rules for Use Case (rules.uc)

Essential story (rules.uc1)
Each Use Case names an essential story about using the system (respectively system features).

Rationale: Other Use Cases are either too detailed, too abstract or not relevant enough, adding them would make understanding the core functionality harder.

Success Criteria: (1) The Use Case names a user-goal which can be traced back (directly or indirectly) to a business requirement stated in Vision&Scope, thus it shows the value a user gets from the system features. OR (2): The Use Case story describes how the system interacts in a not obvious situation, i.e. the Use Case clarifies an important aspect of using the system.

Comment:	A user is the ‚outside world', meaning a human user (a role) or another system/sub-system with an interface to the system.

Interactions (rules.uc2)
Each Use Case describes only the interactions between the world outside and the system necessary to achieve a user benefit. Each step describes the essence of this step, i.e. the intention, not the technical details.

Rationale:	Otherwise not the problem, but a solution is stated, i.e. the Use Cases would be too detailed and there would be too many of them.
Success Criteria:	The system is described as a black box, i.e. each step describes solely an interaction of user and system, but no internal reactions or calculations.
Bad Example:	Step „User authorises himself to the system" is ok; „A password is send by TCP/IP on port 432 to the system encrypted with SSL" is NOT ok

Short story (rules.uc3)
Each Use Case tells a short, simple story about using the system (the primary or „happy day" scenario). All complications (alternatives, exceptions, preconditions, constraints etc.) are explored put in additional sections.

Rationale:	To make the essence clear and understandable.
Success Criteria:	The primary scenario has no branches (e.g. „if"-statements) and there's nothing to extract away from the story.

Generic (rules.uc4)
The generic rules for specification (rules.req) are applied.

2.3.3 Case Study on Defined Abstraction Layers

In the following we will illustrate the concept explained above with a case study from automotive electric/electronic development.

The reader should be aware, that whilst the case study is based on realistic technical assumptions, it is still fictitious. In the first place, this is due to the fact that the levels of abstraction we are discussing here do not exist in our business units. This means, we had to re-engineer them from existing specification documents, which are full of technical details without much consideration of more abstract information. The more abstract and strategic the information presented below gets, the less real it is. For good reasons, especially the vision and scope part is completely fabricated.

As case study we selected the wash/wipe functionality: from the user's point of view this is everything that keeps his front sight free. At DaimlerChrysler, there is an expert for this functionality, who is responsible to provide the corresponding specification documents as well as controlling the component as developed and delivered by an external supplier.

We chose the wash/wipe functionality for the following three reasons:

1. From the driver's point of view, the wash/wipe requirements and capacity are very obvious. Everyone knows what to expect and what a modern car provides.
2. The wash/wipe system is complex enough to prove the benefits of our concept. The specification our case study is based upon is a Word document with some 70 pages. To completely understand all the details – and especially their rationale and legitimacy – is very hard. To roughly understand and document the overall picture takes a highly skilled engineer several days; and this engineer needs to talk to the expert several times in order to get the picture right …
3. The wash/wipe system is a perfect example for showing that in modern automobiles even things that seem very simple at first glance tend to be very complex, as nowadays solutions are very sophisticated.

2.3.3.1 Vision & Scope

Although the information presented in this section should not be mixed up with actual DaimlerChrysler goals and constraints for the windscreen wiper system, we still claim that the following goals for the wash/wipe feature are realistic and give the flavour of what should be in the real document.

First we present the list of goals for the wash/wipe system:

ID	***Vision (Goals)***	***Description***
G-1.1	**Maximise view for driver**	Maximise view in a <appropriate time span> in different [wiper environmental situations].
G-1.2	**Minimise driver distraction**	Minimising user distraction: the higher the wipes per minutes and/or the faster the wiper speed, the higher the user distraction.
G-1.3	**Ease of comfort (handling)**	as little as possible user interactions needed to have correct wiper functionality in different [wiper environmental situations]
G-1.4	**Safety (no harms due to wiper)**	beware user from safety threats and discomfort through washing functionality, i.e. a wash doesn't cause any harm to the driver, the car, passengers or other cars
G-1.5	**Ease of comfort (general)**	if possible, the wash wipe-functionality exploits possibilities to enhance the user comfort in general (support for other car features)
G-1.6	**Support disabled driver**	Disabled driver may need special means for user interaction
G-1.7	**Maximise wiper robustness & life span**	Typical driver (<appropriate mileage>/year in Middle Europe) must not need new wipers (MB originals) before <appropriate time span> years. Average wiper service defects in first<appropriate time span> years: less than 1 per <appropriate number of> cars
G-1.8	**Minimise power consumption**	Wash/wipe must provide appropriate contributions to power management.

		contributions to power management.

Table 2-4: Example goals for the Wash/Wipe system

To document the major goals of the wash/wipe system has two important reasons:

1. When developing and refining the system, the responsible engineer can establish his or her work on a sound fundament. As long as a costly design decision provides a major contribution to, say, minimised power consumption, the engineer has good reasons and arguments to assert himself on the cost department that is trying to save development or production costs by deleting the specified feature. In other words: design decisions and corresponding requirements should contribute to an overall goal, otherwise they are doubtful or even superfluous [Backward Traceability].
2. When trying to evaluate a given specification, management is able to check whether and how each goal is considered in the system developed. If there is no feature or requirement that deals with minimising power consumption, nobody is able to claim to do so. In other words: each goal should be refined and end up in some concrete requirements that make sure the goal gets fulfilled [Forward Traceability].

From the EMPRESS point of view – managing the evolution of embedded systems – the usefulness of the system goals and their correlation to design decisions and system requirements are obvious: Reuse of requirements artefacts can start with comparing present goals with recent ones.

Besides the goals, there are also constraints on the set of possible solutions (scope) and on the development process. Our case study list is the following:

ID	***Scope***	***Description***
S-2.1	**Deployment of Model Y components**	Wash/wipe Model X is to be developed using existing components of Model Y (two wiper system, sys-id 0815); for Model X we do not want to develop new hardware and mechanics, but reuse existing technology.
S-2.2	**Rain sensor feature**	The rain sensor feature as prototyped by advance development shall be introduced as important innovation.
S-2.3	**Development process**	Wash/wipe component is to be developed according to [Dev Process Model-X, standard component with late call for proposal] specifying exact milestones and quality gates.
S-2.4	**Rain sensor stakeholders**	Integration of rain sensor shall be carried out in close cooperation with Hans Dampf from Marketing and Daniel Düsentrieb from advance development. Hans Dampf is to approve all user requirements, Daniel Düsentrieb is to approve both all user requirements and all system requirements.
S-2.5	**Wash/wipe sponsor**	Dr. Hutzliputzli is responsible for electric/electronics in Model X and takes or approves any final decision.

Table 2-5: Example scope for the Wash/Wipe system

From this second list we only want to discuss object S-2.2, the rain sensor feature. This entry exemplifies a situation typical for Mercedes-Benz, but probably typical for many automotive companies. At Mercedes-Benz, development is technology-driven. Our engineers are famous for and keen on inventing new features. Mercedes-Benz management asks for innovations, engineers invent new features. With respect to the concept we are discussing here, this means that the specification process is a feature-driven one. This is the reason we are going to discuss below the feature list before the Use Case model.

2.3.3.2 Feature List

In our case study the wash/wipe system is represented by the following features list.

ID	***Feature***	***Description***	***Goals***
F-001	**Interval Wipe**	wiper is cyclically activated for singular wipe moves every <appropriate number of> seconds.	G-1.1, G-1.2, G-1.3, G-1.7, G-1.8
F-002	**2 Fixed Wiping Speeds**	there are two fixed wiping speeds.	G-1.1, G-1.2, G-1.3, G-1.7, G-1.8
F-003	**Pulse Wipe**	wiper carries out one wipe move.	G-1.1, G-1.2, G-1.3, G-1.7, G-1.8
F-004	**Rain Sensor (Adaptive Wiping)**	system automatically controls start of wiping cycle and/or wiping speed according to the intensity of rain.	G-1.1, G-1.2, G-1.3, G-1.6, G-1.7, G-1.8, S-2.2
F-005	**XXX**	< Feature deleted as not to be published >	G-1.5
F-006	**Wipestart Suppression**	opened doors or bonnet suppress wiper cycle (if (a) interval wipe or rain sensor wipe and (b) car speed below <appropriate small speed>.	G-1.3, G-1.4, G-1.5
F-007	**Automatic Wiping Speed Reduction**	high wiping speeds are automatically reduced when car slows down below some <appropriate speed>.	G-1.2, G-1.3, G-1.7, G-1.8
F-008	**Finish Wipe**	when wiping is stopped, wiper completes current wipe cycle in order to return to parking position.	G-1.1, G-1.2, G-1.3
F-009	**Wash**	water pump shoots wash fluid on windscreen and wipers clean windscreen.	G-1.1, G-1.3
F-010	**Wash Post-Wipe**	"wash" procedure is always completed by additional wiper cycles to make sure windscreen is dry.	G-1.3
F-011	**XXX**	< Feature deleted as not to be published >	G-1.1, G-1.3, G-1.7
F-012	**Block Protection and Detection**	system detects stuck wiper and reacts in an appropriate way.	G-1.7, G-1.8
F-013	**Disabled Driver Swith**	disabled driver is able to operate (specific subset of) wiper functionality	G-1.6
F-014	**XXX**	< Feature deleted as not to be published >	G-1.7, G-1.8

Table 2-6: Example features for the Wash/Wipe system

For each feature the “Goals” column represents the rationale relation as presented in Figure

2-11 and Table 2-4). In other words: each feature contributes to all the goals from vision and scope that are listed in the corresponding column. From the top down point of view, it is important to note that each goal is contributed to by at least one feature (forward traceability) – so there is still hope that each goal will be taken care of. From the bottom up point of view, every feature contributes to at least one goal (backward traceability) – there is no feature without reason. Obviously, on this level we could discuss whether we could reach goal G-1.1 without feature F-002, for instance, as there are enough features left to take care of this goal. (Working for Mercedes-Benz, this discussion is not taken into account …)

In requirements engineering textbooks it is easy to separate problem and solution space. In problem space the system analyst describes the problem the system to be built is going to solve, whereas in solution space the developer describes how the system is going to solve the given problem. In real specification documents this separation is hard or even impossible to achieve. In reality with lots of sub-systems built by different suppliers problem and solution are very intertwined.

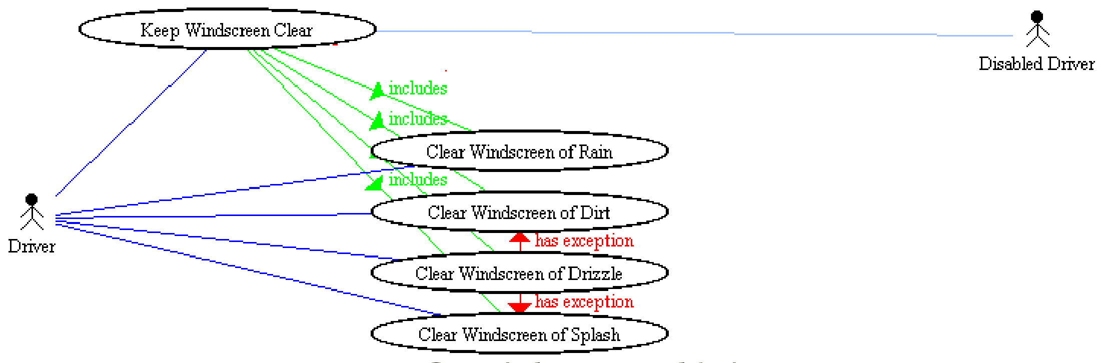
On the feature level, however, the separation between problem and solution works quite well. Both the features and the Use Cases (see below) are on the same level of abstraction (see Figure 2-11), the main difference being that features are in solution space and Use Cases are in problem space. Features describe technologies and partly solutions the overall system is going to offer.

2.3.3.3 Use Case Model

In the Use Case model we describe the problem we are going to solve by the new system built of the features listed above. For wash/wipe this means, that we document the benefits and advantages of wash/wipe from the driver's or passenger's point of view. Whereas features like rain sensor are static technical entities in solution space, Use Cases are dynamic courses of events in problem space. Whereas features are more or less fragments the solution is put together of, Use Cases represent the glue and describe the appropriate co-operation between those features.

In order to describe the problem the new system is going to solve, Use Cases seem perfect. The number of Use Cases to completely and coherently describe a typical automotive electronic component is not very high. For wash/wipe, for instance, we are dealing with five of them:

1. Keep Windscreen Clear
2. Clear Windscreen of Rain
3. Clear Windscreen of Dirt
4. Clear Windscreen of Drizzle
5. Clear Windscreen of Splash



Use Cases for 'Windscreen Wash/Wipe'

Figure 2-14: Use Case Model for wash/wipe function

Figure 2-14 shows an important property of a Use Case model (see e.g. [AZ02]): by inclusion (include relation indicated by green arrow) we are able to defer complexity to levels below. Formally, the wash/wipe Use Case model contains one basic Use Case “Keep Windscreen Clear”. This is the most important job the system is built for. Taking a closer look at this task, we find that it makes sense to distinguish between several external conditions, namely rain, dirt, drizzle, and splash. But we can describe those special cases of the main task later, namely in the corresponding Use Cases.

The top level Use Case “Keep Windscreen Clear” looks like this:

Used Features: F-001, F-002, F-003, F-004, F-005, F-006, F-007, F-008, F-009, F-010, F-011, F-012, F-013, F-014
Actor: Disabled Driver
Primary Actor: Driver

2.3.3.3.1 Primary Scenario

UC-16 It starts raining.

UC-163 Driver selects Clear Windscreen of Rain. The wipers keep the windscreen clean by wiping at a speed appropriate to the intensity of the rain.

includes Clear Windscreen of Rain

UC-164 Driver switches off windscreen wiping. The wiper arm returns to its Parked position.

2.3.3.3.2 Alternative Paths

UC-17 It is drizzling or raining lightly. Driver selects Clear Windscreen of Drizzle. The car wipes the windscreen intermittently according to the build-up of water on the windscreen. Driver switches off windscreen wiping.

includes Clear Windscreen of Drizzle

UC-165 It is raining very lightly. Driver selects Clear Windscreen of Splash. The car wipes the windscreen once.

includes Clear Windscreen of Splash

UC-279 The windscreen is dirty. Driver selects Clear Windscreen of Dirt. The car pumps water on to the windscreen, and wipes it to remove the pumped water and the dirt.

includes Clear Windscreen of Dirt

UC-272 Disabled Driver selects wash/wipe commands using the disabled person’s switch. The car responds exactly as to normal wash/wipe commands. Where a command is received from both sources, the normal (Column Switch) command takes priority.

2.3.3.3.3 Exceptions

UC-18 Wipers stuck: wiper motor times out after a fixed period, and informs driver that wipe has failed. Wiping is only resumed when the driver gives a fresh wiping command (using the Column Switch).

UC-201 Out of wash fluid: car informs driver that it is out of wash fluid.

UC-202 Windscreen covered in snow or ice: driver is responsible for clearing the windscreen.

UC-218 Bonnet opened: [Proposed: wash /] wipe stops. The current wipe cycle is completed to allow the Wiper Arm to return to its Parked position. When the bonnet is closed, wiping resumes.

UC-219 Driver’s and/or Front Passenger’s door opened: [Proposed: wash /] wipe stops. The current wipe cycle is completed to allow the Wiper Arm to return to its Parked position. When the door is closed, wiping resumes. (Unclear whether this applies only to Interval (Drizzle) wiping; it seems to be needed in all cases.)

UC-261 Ignition switched off: wash/wipe stops. When the driver switches the ignition on again, the wiper arm returns to its Parked position.

2.3.3.3.4 Constraints

UC-19 Materials in the wash fluid, such as anti-freeze, de-icer, and detergent, are permitted by EU regulations for automotive use.

UC-211 Materials in the wash fluid do not damage the types of paint, rubber and glass used in the car.

UC-213 Engine operation is unaffected by windscreen cleaning.

2.3.3.3.5 Trigger

UC-21 Driver selects one of the Wash-Wipe commands.

2.3.3.3.6 Preconditions

UC-23 Windscreen is not covered in snow or ice.

UC-203 Ignition in Position 1 (Radio Position).

UC-204 Driver's and Front Passenger's doors closed.

UC-205 Bonnet closed.

UC-206 Battery voltage is above the wash/wipe threshold (8 volts).

2.3.3.3.7 Stakeholders and Interests

UC-25 Driver needs to see clearly through the windscreen, so as to drive safely.

UC-178 The Law and the Police require Driver to be able to see properly at all times.

UC-173 Driver and Passengers want not to get wet.

UC-212 Driver wants engine operation to be unaffected by windscreen cleaning.

UC-214 Mechanic wants windscreen cleaning system to be simple to diagnose and repair.

2.3.3.3.8 Minimal Guarantees

UC-27 Driver and Passengers are not wetted.

UC-174 Battery is not drained when ignition is not switched on.

UC-207 Wipers are not scraped over a dry windscreen.

2.3.3.3.9 Success Guarantees

UC-29 Windscreen is cleared.

This is not the place to have a complete introduction to Use Cases, but there are some properties we would like to clarify on the given example:

- The name of a Use Case normally names a goal the system described is supposed to achieve.
- Each Use Case is built of several so called *scenarios*. A Use Case is a collection of scenarios that belong together for semantic reasons. A scenario is one possible routine of events.
- There is a so called *primary scenario* describing the "happy day" scenario, the normal course of events.
- There are *alternative paths* describing possible alternative flow of events. In the example we differentiate between several outside conditions. Normally, each alternative is represented by a corresponding scenario; in our example three alternative scenarios are refined in individual Use Cases.
- There are *exceptions* that deal with system failure or conditions the system cannot cope with in order to fulfil the Use Case goal. Again, each exception is specified by a corresponding scenario that can be refined by another Use Case, if necessary.
- Then there is some other formal stuff, as needed, like constraints, trigger, preconditions, stakeholders. To this end, the example is self explanatory.
- For our overall concept, the first line entry "used features" is important: Here we list all the features that contribute to the Use Case. This entry represents the "uses" relation established from Use Cases to feature (see Figure 2-11).

In order to further exemplify the refinement of uses cases by the include relation here we present just the main ingredients (primary scenario, alternative paths, and exceptions) of the "Clear Windscreen of Rain" Use Case, which is the first Use Case included in the "Keep Windscreen Clear" Use Case.

Used Features: F-002, F-004, F-008
Primary Actor: Driver
is included in Keep Windscreen Clear

2.3.3.3.10 Primary Scenario

UC-183 The driver manually specifies the wiping speed (Slow or Fast). The car wipes the windscreen at that speed.

UC-254 The driver manually switches windscreen wiping off. The car completes the current wiping cycle, returning the wiper arm to its parked position.

2.3.3.3.11 Alternative Paths

2.3.3.3.11.1 UC-255 Automatic Rain Clearance: The car measures the intensity of the rain and wipes the windscreen at the speed that best suits the intensity of the rain.

2.3.3.3.12 Exceptions

UC-186 As for 'Keep Windscreen Clear'.

UC-247 Driver selects 'Clear Windscreen of Splash': the car continues the wiping sequence of 'Clear Windscreen of Rain' as if nothing had happened.

UC-267 Wiper stuck at wiper speed II for time-out period: wiper motor is reset to speed 1 and wiping is again attempted for the fixed time-out period. Should this also fail, the motor is switched off and driver is informed.

UC-268 Wiper stuck at wiper speed 1 for time-out period: wiper motor is switched off and driver is informed.

UC-274 Voltage below Wiping threshold and Wiper speed II selected and no signal from Wiper Cam for time-out period: Speed II is disabled.

In practice it makes sense to further refine the “uses” relation from Use Cases to features (see Figure 2-11) by attributing the very scenario that actually makes use of the linked feature. For instance, in the “Clear Windscreen of Rain” the “Rain Sensor” feature F-004 is actually used in the alternative path described by the scenario UC-255 “Automatic Rain Clearance”.

2.3.3.4 Test Cases

In our model test cases are derived from Use Cases and specified at a high (user) level. Thereby we want to ensure that every relevant user interaction (and thus the whole system requirements) will be tested. For every section of a Use Case in which a software error could occur at least one test case has to be specified. Thus a test case is not explicitly linked with single system requirements but, as every relevant Use Case path will be tested and every system requirement can be assigned to a Use Case, also every system requirement will be verified by a test.

Of course, the high level test cases have to be made more concrete by low level test cases later, but this step is not really problematic any more, as the completeness of the test cases has been already ensured and the high level test cases just have to be described on a lower level.

The following table shows example test cases for the Use Case “Keep Windscreen Clear”. In the “Use Case” column the number of the Use Case section that makes the test case necessary is noted.

ID	Test Cases	Preconditions	Input Data	Postconditions	Expected Result	Result	Use Case
TC-28	Driver switches off wiping	Wiping active	Driver selects wash / wipe to stop	Wiping inactive	Wiper arm returns to its parked position		UC-164
TC-14	Wipers stuck before wipe command	Wiping inactive, Wipers stuck	Driver selects wash / wipe	Wiping inactive	Wiper motor stops after x seconds, driver is informed	Ok	UC-18

ID	Test Cases	Preconditions	Input Data	Postconditions	Expected Result	Result	Use Case
TC-15	Wipers stuck during wiping	Wiping active, Wipers stuck	Wiper gets stuck	Wiping inactive	Wiper motor stops after x seconds, driver is informed		UC-18
TC-16	Out of wash fluid	No more wash fluid available	Driver selects wash / wipe	Wiping inactive	Wiper arm returns to it's parked position, driver is informed	Wiper wipes without wash fluid	UC-201
TC-32	Bonnet is open before wipe command	Wiping inactive, bonnet open	Driver selects wash / wipe	Wiping inactive	Wiping commands are not obeyed, driver is informed		UC-218
TC-33	Bonnet is opened during wiping	Wiping active	Bonnet is opened	Wiping blocked by open bonnet	Wash / Wipe stops, wiper arm returns to its parked position, driver is informed		UC-218
TC-34	Bonnet is closed during blocked wiping	Wiping blocked by open bonnet	Bonnet is closed	Wiping active	Wiping commands that have been chosen before the bonnet was opened are obeyed, driver is informed		UC-218
TC-35	Wiping is turned off after wiping had been blocked by bonnet	Wiping blocked by open bonnet	Wiping turned off	Wiping inactive	Wiper doesn't change it's position		UC-218
TC-36	Bonnet is closed during inactive wiping	Bonnet is open, wiping inactive	Bonnet is closed	Wiping inactive	Wiper doesn't change it's position		UC-218
TC-37	Front door opens during wiping	Wiping active, all front doors shut	a front door is opened	Wiping blocked by front door	Wiper arm returns to it's parked position, driver is informed		UC-219
TC-38	One front door shuts after wiping had been blocked by front door	Wiping blocked by front door, one front door shut	second front door shuts	Wiping active	Wiping commands that have been chosen before the wiping was blocked are obeyed, driver is informed		UC-219
TC-39	Both front doors shut after wiping had been blocked by front door	Wiping blocked by front door, both front doors open	both front doors shut at the same time	Wiping active	Wiping commands that have been chosen before the wiping was blocked are obeyed, driver		UC-219

ID	Test Cases	Preconditions	Input Data	Postconditions	Expected Result	Result	Use Case
					is informed		
TC-40	Wiping is turned off while wiping blocked by front door	Wiping blocked by front door	Wiping control is turned off	Wiping inactive	Wiper doesn’t change it’s position		UC-219
TC-41	Ignition switched off during wiping	Ignition switched on, Wiping active	Ignition switched off	Wiping inactive, Wiper unparked	Wiper stops at it’s current position		UC-261
TC-44	Ignition switched on when wiper arm is unparked and wash/wipe switched on	Ignition switched off, wiper unparked, wash/wipe switched on	Ignition switched on	Wiping active	Wiper arm returns to is parked position and then wipes again		UC-261
TC-42	Ignition switched on when wiper arm is unparked and wash/wipe switched off	Ignition switched off, wiper unparked, wash/wipe switched off	Ignition switched on	Wiper parked, Wiping inactive	Wiper arm returns to it’s parked position and stays there		UC-261

2.3.3.5 System Requirements

A typical (functional) system requirement looks like the following example from wash/wipe:

name	Wipestart Suppression, Door Contact
ID	FSS-171
inputs	… [interfaces to several related components] …
outputs	… [interfaces to several related components] …
precondition	None.
functional prerequisite	… [lots of technical details deleted] car speed below [parameter v_{WLLU}] signal “contact registered” from door control wash/wipe in mode “finish wipe” or “interval wipe”
trigger	RLS requests „wipe“ cycle
content	If RLS requests wipe cycle whilst door control reports opened door and whilst the car speed is below [parameter v_{WLLU}], the wipe cycle request is suppressed until either both doors are closed or car speed increases above [parameter v_{WLLU}]. A wipe cycle that has already been started is not immediately stopped, but finished completed until the wiper enters the parking position.
feature rationale	F-001 (Interval Wipe) F-004 (Rain Sensor) F-006 (Wipestart Suppression)
Use Case rationale	UC-49 (Clear Windscreen of Drizzle) UC-219 (Driver’s and/or Passenger’s Door Opened) / Exception of UC-11 (Keep Windscreen Clear) UC-255 (Adaptive Rain Clearance) / Alternative of UC-179 (Clear Windscreen of Rain)
comments	... [open questions, additional explanations]
author	... [author name]
state	to be reviewed

Like Use Cases, system requirements are to be documented in some kind of template asking for several kinds of information.

The system engineer is supposed to derive system requirements from features and Use Cases. The other way round, the engineer is supposed to document the derivation by establishing links between system requirements and features as well as between system requirements and Use Cases. In our feature driven approach, we claim the link between system requirements and features as mandatory, whereas the link between system requirements and Use Cases is optional, but highly recommended (see corresponding “Rationale” relations in Figure 2-11). The reason for this recommendation is that by establishing both a rationale between system requirements and Use Cases (n:m) as well as between test cases and Use Cases (n:1), we implicitly establish the important relation between system requirements and test cases (n:m).

2.3.3.6 Complete Picture

In sections 2.3.3.1 “Vision & Scope” to 2.3.3.5 “System Requirements” we presented and discussed our approach to specifying embedded systems in automotive industry in order to cope with evolution. In accordance to the real development process as proposed by our concept, the introduction and discussion of the several levels was top down, from high level and strategic decisions down to technical details of the concrete solution.

To complete the picture, in this section we briefly re-sketch the overall picture, this time bottom up starting from the low level system requirement from section 2.3.3.5, namely “Wipestart Suppression, Door Contact”.

System requirement FSS-171 “Wipestart Suppression, Door Contact” specifies the exact conditions and detailed routine of suppressing wipe cycles in order to make sure that passengers entering or leaving the car don’t get wet.

The rationale for FSS-171 with respect to Use Cases and scenarios (optional relation) is threefold: FSS-171 contributes to

1. Clear Windscreen of Drizzle (UC-49), as wipestart suppression is needed in interval wipe mode.
2. Automatic Rain Clearance (UC-255), as wipestart suppression is needed in adaptive wipe mode.
3. Driver’s and/or Passenger’s Door Opened (UC-219), which is the direct abstract specification of FSS-171

Whereas the rationale relation from FSS-171 to UC-219 is obvious, the relation to UC-49 and UC-255 is not obvious in the first place. But by analysing the wipestart suppression idea we find that wipestart suppression makes sense in interval wipe and adaptive wipe mode.

The rationale for FSS-171 with respect to features (mandatory relation) is threefold, as well: FSS-171 contributes to features

1. Interval Wipe (F-001),
2. Rain Sensor (F-004), and
3. Wipestart Suppression (F-006).

Feature Wipestart Suppression (F-006) rationale covers

1. Ease of Comfort (handling) (G-1.3),
2. Safety (no harms due to wiper) (G-1.4), and
3. Ease of Comfort (general) (G-1.5),

which are the goals FSS-171 offers a main contribution to.

Exploiting the (optional) rationale relation from test cases to Use Cases, for the use UC-219 we identify the corresponding test cases

1. Front door opens during wiping (TC-37),
2. One front door shuts after wiping had been blocked by front door (TC-38),

3. Both front doors shut after wiping had been blocked by front door (TC-39), and
4. Wiping is turned off while wiping blocked by front door (TC-40).

Consequently, these test cases are part of the test suite to make sure that the final wash/wipe system is correct with respect to wipestart suppression.

2.3.4 Tailoring of the Abstraction Layer Model

By the application of this approach in real projects, we learned that the three layers as described in the sections above are sometimes insufficient as it is hard to allocate all relevant informations clearly to a particular layer. As a consequence, we were forced to tailor the generic approach towards the needs of an individual project.

This paragraph illustrates a possible tailoring of the abstraction layer model, which has been used in detail for the specification of the instrument cluster in workpackage 5.

The basic model differentiates three types of requirements. Business requirements, user requirements and system requirements. In our revised model we have four layers for these three sections. Figure 2-15 illustrates the mapping of the recent sections and the new layers. Between the layers there are many interconnections which are later shown in Figure 2-16.

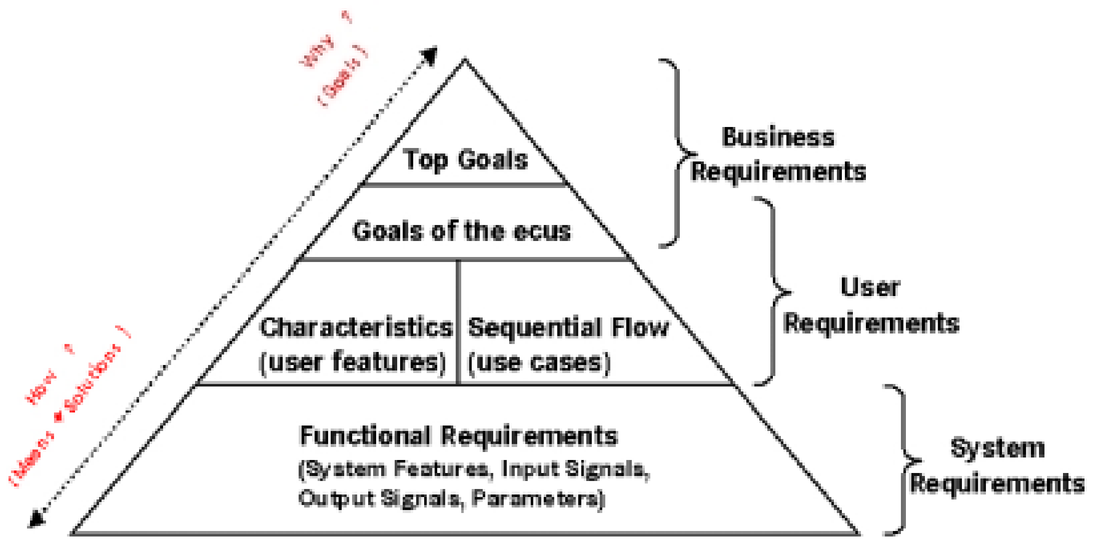


Figure 2-15: Tailored Abstract Layer Model

In the instrument cluster project we refined the business requirements layer into top goals and goals. The idea behind this differentiation is that there are goals, which are valid for all subsystems inside a car and goals that are particular to the instrument cluster. The top goals form the framework for all subsystem goals and can be seen as the brand policy of a company. The top goals are refined by goals that are directly derived from top goals. Additionally they cover user intentions that are more elaborated by means of Use Cases. Every goal can by refined into numerous features. A feature is a special attribute of the subsystem and implements one or more aspects of its goals. Features also show up as part of the Use Cases. Every feature must appear at least in one Use Case. Features may be seen as a static implementation of goals whereas Use Cases describe their dynamic instantiation.

We furthermore enhance the system requirements layer by a context diagram according to SA/RT [HP88, Par98]. The context diagram gives a figurative overview of the specified subsystem. There the system itself is drawn as an eclipse with attached rectangles that represent adjacent subsystems. Additionally the context diagram shows the information flow between the subsystems. Thus it defines all input and output signals of the specified subsystem.

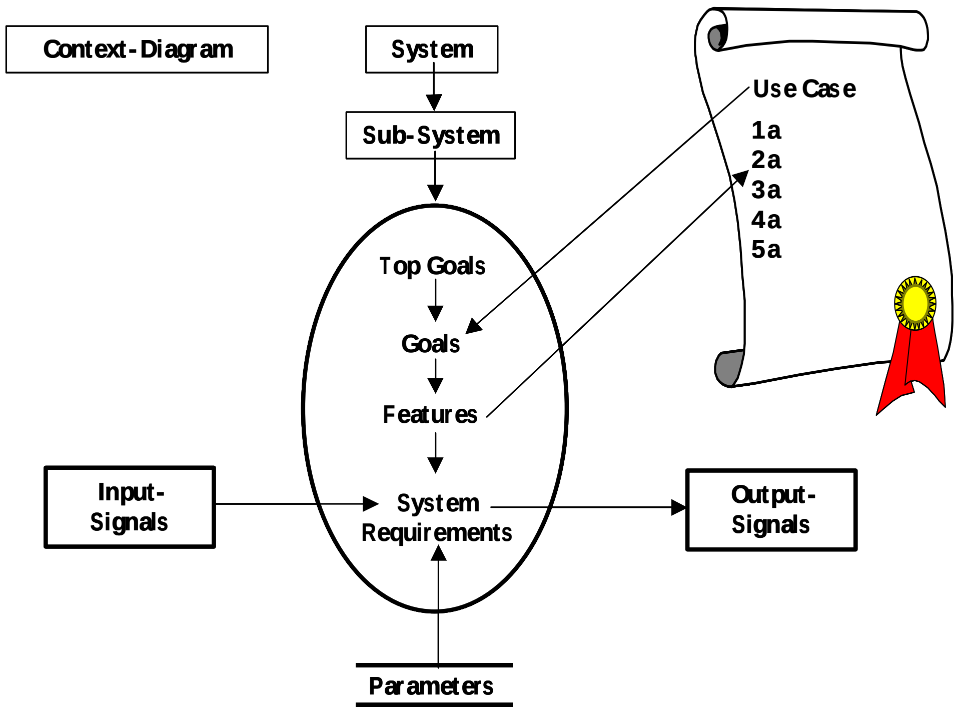


Figure 2-16: Link Structure Model

Figure 2-16 visualises the link structure between the various parts of our tailored abstraction layer model. It is a conclusion of some experiments. There we identified the following linking rules:

- Every top goal has outgoing links into each subsystem
- Every goal has an incoming link from a top goal
- Every goal has an incoming link from a (customer) Use Case name
- Every feature has an incoming link from a goal
- Every feature has an outgoing link into the (customer) Use Cases
- Every feature has an outgoing link to the system requirements
- Every part of the system requirements has an incoming link from a feature
- Every part of the system requirements has an incoming link from an input signal (optional)
- Every part of the system requirements has an incoming link from a parameter (optional)
- Every part of the system requirements has an outgoing link to an output signal (optional)

By applying these rules while creating a system specification, we mentioned that the system specification gets on the one hand a clear and understandable structure and on the other hand it gets easier to integrate new requirements into the right sections. Obscurities about new features can be eliminated in a consistent manner.

2.3.5 Dealing with Evolution

2.3.5.1 Definition of Evolution and its Context

In this section we give an overview of the context, in which our evolution process approach can be applied. We define evolution and other important terms.

The idea of evolution is a central one within the project EMPRESS. In the context of this project we may characterise evolution of systems as change in a general sense. There might be numerous reasons behind these changes. We may observe shortcomings of an existing product compared to user's needs, but we may also observe new needs (which might result in an enhanced product). Evolution causes the creation of a new product, which is either a replacement of an existing product or which results in a new product within the same product family (see [NoC] and the definition below). We do not use the term evolution if we want to talk about features emerging during system development or changes applied to a final product, which are used to fix (obvious) bugs.

Clearly, there is a close connection between evolution and (formal) change management. We define formal change management as a process that formalises tracking and implementation of individual change proposals, usually driven by a state model ranging from "initiated" to "released".

In principle, we can use the concept of formal change requests not only for bug fixing, but we can also use it to specify new products within the same product family by means on change requests. However, this approach is quite unusual. It would be more common to say "the new system should be like xyz with the following modifications". Thus we define which parts of the existing system are parts to be reused (at least on the specification level) and which parts have to be modified, enhanced, or deleted.

Having the idea of product families in mind, we have to define precisely what the scope of a product definition is. At the first glance we may only think about specifications, which define exactly one product. Given the necessity to specify several systems that evolve over time and form a complete family of systems, it might also make perfect sense to maintain the specifications for the individual products not as individual artefacts but as one "multi-product" specification with the necessary variations embodied (see Figure 2-17).

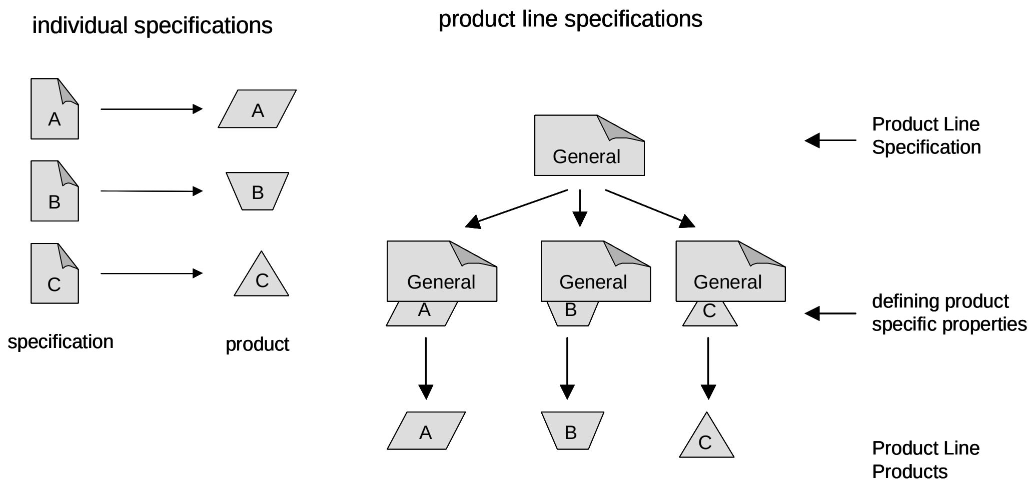

individual specifications
product line specifications
A
B
C
A
B
C
specification
product
General
General A
General B
General C
A
B
C
Product Line Specification
defining product specific properties
Product Line Products

Figure 2-17: Specifications for families of products.

And even if we plan to consider only one product, anyhow, we in fact often deal with a whole set of products. For instance, if we specify one electronic control unit (ECU) like the instrument cluster for the new XYZ-class, we in fact specify a set of instrument clusters (which might differ in the optical design, used scales [i.e. mph or km/h] and so on). Sometimes these differences can just be implemented by means of configuration parameters; in other cases we really observe physically different products. Usually it does not make any sense to maintain separate specification artefacts for each product variant. To clarify the different views, we use the following terms in the remainder of this chapter:

- **Product family** as a collection of individual prime products that share some common functionalities and features. The members of the product family are built by individual (but usually related) development projects.
- **Product line** as a collection of individual prime products that share a common, managed set of functionalities and features and which are developed in a prescribed way. (see [NoC])
- **Prime products** are the result of one system development project.
- Within a prime product development project we might deal with several **product variants**, i.e. specific instantiations of the prime product. The instantiation might be realised merely by means of parameterisation, but it might also require different hardware implementations. It is also likely to have a tree of product variants where the higher variations are implemented by means of hardware whereas the variations towards the leaves are implemented by parameterisation.

Figure 2-18 illustrates the interplay of product families, product lines, prime products and product variants graphically by using the example of instrument clusters for vehicles.

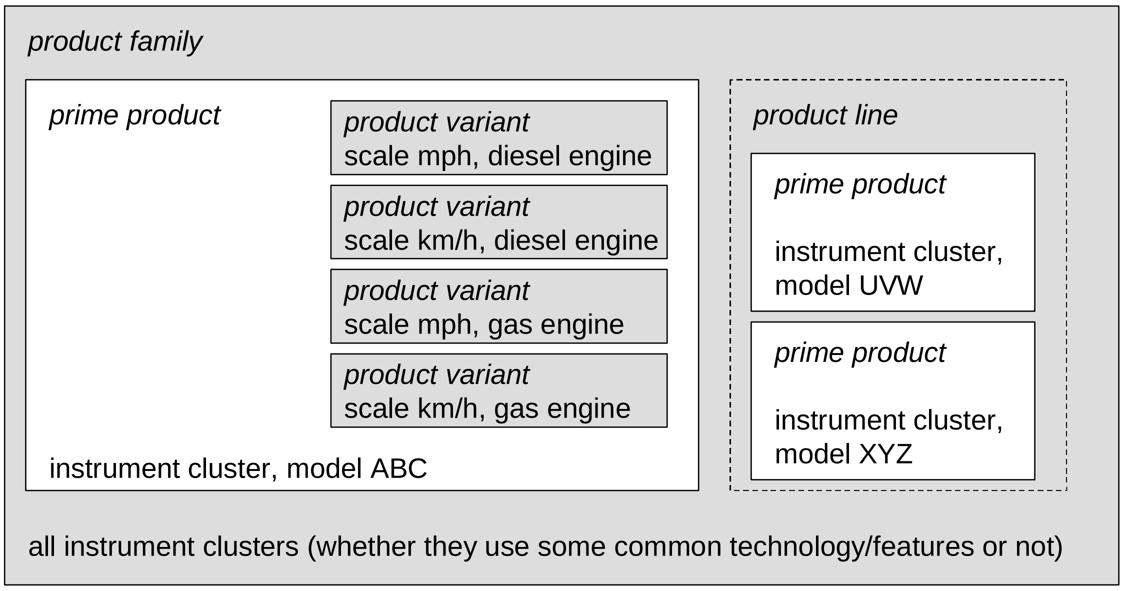

product family
prime product
product variant
scale mph, diesel engine
product variant
scale km/h, diesel engine
product variant
scale mph, gas engine
product variant
scale km/h, gas engine
instrument cluster, model ABC
product line
prime product
instrument cluster, model UVW
prime product
instrument cluster, model XYZ
all instrument clusters (whether they use some common technology/features or not)

Figure 2-18: Product families, product lines, prime products and product variants.

In the context of this chapter, we look primarily at product family specifications (i.e. specifications that define a set of products rather than one product). But, as we will see, most of these ideas are applicable to the single product specification case, too. Especially the reflections on variants are quite intuitive and by that it is easy to integrate them into current projects working with product-specific documentation. We considered the “product-spanned” specification more interesting, because of the possibility to develop a “creeping” product line, which is reality-driven and ongoing.

In the following sections, we present an approach that supports evolution processes already in early phases of the system’s development process. Obviously, we base our approach on the concept of abstraction layers as introduced in Section 2.3.2. While doing so, we put special emphasis on handling of variations inside specifications. With that we provide a possibility to define product lines in a bottom-up manner. Finally, we illustrate our findings by means of two case studies in 1.1.6 “Case Study on Evolving Specifications”.

2.3.5.2 Evolution Processes and Impact Analysis

In this section, we look at evolution processes, which a (requirements) engineer faces nowadays, and means for impact analysis in the context of evolution. After a description how the abstraction layer approach can be used to support the process of evolution, we figure out the most important points in the context of an impact analysis with an instruction of application.

The abstraction layer approach, as introduced in Section 2.3.2, supports the identification of the consequences of changes and evolution. A change at layer X usually causes iteratively change(s) at layer X+1. The parts in layer X+1 that have to be changed can be found very easily. The root-change-point at layer X refers to all possibly concerned parts at layer X+1, such that the search space for the change becomes quite small and manageable. This search space has to be adjusted by hand meaning that only by the knowledge of the (requirements) engineer the impact of the change/evolution can be found out. But nevertheless, the structured filtering of possibly concerned requirements helps a lot – compared to the actual approach, which is to go through the whole specification to find out modification candidates. A change at the lower levels of the abstraction layers (User Requirements and System Requirements) does not necessarily have an impact on a higher layer. In the case an impact was identified, the change at the higher layer is performed and it is continued as described above.

Clearly, this approach is not new. Similar ideas can be found in many publications (see, e.g., [SBJA]). The crucial part here is to base those tracing information on meaningful elements. Here, the introduced abstraction layers provide a good foundation.

Obviously, complete references between the layers are an inevitable precondition for the applicability of this approach. This means that all abstraction layers have to be filled completely in a top-down manner. Thus, the set of documented goals have to reflect both the business point of view and the user or system requirements point of view. This means that no important goal from a business perspective is missing and no requirement on a lower level fulfils a goal which is not mentioned explicitly. Furthermore Use Cases and features (i.e. user requirements) have to refine the business goals providing a functional description of the system, and each system requirement is needed to fulfil requirement(s) from a higher layer. The links between the abstraction layers have to be complete, consistent, and ongoing. Last but not least the whole specification has to be managed and kept current (even over the completion of the development of the system) to have a reliable basis.

To our experience, it is very hard to maintain bi-directional link information explicitly, i.e. to maintain explicit “Links” between, for instance, Use Cases and business goals. It is much easier to refer from each requirement to its origin in the sense that the origin is the justification for the existence of the individual requirement. Given these upward-references, however, the corresponding downwards-references can be derived from them automatically.

Example: *Let us apply the change request* “After suspension of wipe suppression there shall be at least two wiping cycles to clean the windscreen properly”. *This change request does not change the get of goal. Instead, it can be seen as a contribution to goal G-1.1 and deals with functionalities imposed by goal G-1.4. This information leads us (beside others) to feature F-006 (via refinement of G-1.4 to F-006). We also reach Use Case “Keep windscreen clear” which refers in its exceptions (UC-219) to wipe suppression. This again directs us to the system requirement “Wipestart Suppression, Door Contact” which we ultimately have to modify. The new content could be like the following:*

> *“If RLS requests wipe cycle whilst door control reports opened door and whilst the car speed is below [parameter v_{WLLU}], the wipe cycle request is suppressed until either both doors are closed or car speed increases above [parameter v_{WLLU}]. A wipe cycle that has already been started is not immediately stopped, but finished completely until the wiper enters the parking position. After the condition for wipe cycle suppression has been become*

invalid, two wipe cycles are initiated. After that wiping activation is done according to the current setting."

As a rule, further implications may result from a change that cannot be captured by the approach described above. A change might cause the necessity for new or completely different requirements as new system aspects are introduced by the change request. Here the basic impact analysis process is quite natural, too. We start by questioning the underlying (business) goal. This causes the modification of existing goals. Then we have a look at the affected features and Use Cases. While doing so, we have to think about interrelations between existing features and Use Case steps. After that we have to take care of the system requirements.

In reality, we can often observe a complete change path, i.e. a sequence of changes, which sometimes end up with the solution that was in place when the change sequence started. The next example illustrates this problem.

Example: *In modern vehicles, steering wheel buttons are quite common. However, these buttons are quite expensive and there is the temptation to remove these buttons. Let us assume an ongoing development project and currently steering wheel buttons are part of the vehicle definition. Now there is the change request to get rid of these buttons. Beside their use in other functions, the buttons were used to reset the short distance counter. As a consequence of the change request, we have to think about another way for resetting the counter. One proposed solution might be to add a single button to the instrument cluster, which is located in its middle. This solution might cause another change request since buttons like these are sub-optimal from an ergonomic point of view (one has to press the button by moving the arm through the steering wheel). The next proposed solution might be to place the button at the side of the instrument cluster. According to mechanical design constraints there is no space for this button. Thus there is the change request to introduce steering wheel buttons and the journey ends at its starting point.*

In reality, such change sequences last for several weeks or months, as each proposed solution is refined to a certain degree until the problems become clear. To avoid such cycles, two strategies have to be applied:

1) Documentation of change paths and design decisions (rationales).

2) Upfront analysis of solution space.

Strategy (1) means that we have to keep a record of all changes applied to the specification. This is clearly good practice in each mature system or software development environment. The implication, however, is that we have to make the change records available nearby the specification such that the requirements engineer is aware of past changes and their rationales. If the change log is just stored somewhere in the depth of a repository, it is unlikely that they are taken into account when a new change request occurs.

Strategy (2) suggests that alternative solutions are explored in a breadth-first instead of a depth-first manner. This means that all solutions, which seem to be feasible are documented and elaborated only to a degree that a meaningful selection could be made. Certainly, exploring and judging solutions is a natural activity for every engineer. However, this process is carried out usually very quick and the intermediate results, i.e. the alternatives and their evaluation, are not documented at all. The same is true for the decision criteria applied to select one alternative.

2.3.5.3 Variation Points

As laid out earlier, we rarely deal with real single product specifications. Instead we either face prime product specifications, which contain several product variants or we see a real product line specification. Now there is the question how to handle such specifications in daily life. We propose to use "variation points", which are embedded in the specifications as an appropriate means for this challenge. In this section we introduce the concept of variation points and describe the procedures to deal with them. Additionally, we give some brief examples and discuss their benefits and limitations. Some more elaborated examples can be found in Section 2.3.6.

In its general meaning, a variation point (in the context of product families) gives the opportunity or necessity to select a specific solution out of a space of given solutions. Some variation points may be related to other variation points and decisions made at one variation point may influence the possible decisions at other variation points.

Another central concept which comes along with variation points, is binding time at which decisions with respect to the various variation points are made. Binding time in general may range from specification time, design time, implementation time, and delivery time to run-time.

In our context, a variation point is an explicit element within a requirements specification that contains five information items:

a. Decision criteria, i.e. some criteria that refer to some (external) constraints or decisions.
b. Alternatives, i.e. a list of constraint values.
c. Solutions, i.e. for each alternative some requirements that are relevant if the corresponding alternative is selected.
d. Selected alternatives (or, if binding is postponed, the corresponding remark)
e. Dependent variation points, i.e. variation points, which have to be considered when making the decision for this variation point.

Figure 2-19 illustrates this by means of an example. Please note that the selected notation uses explicit phrases to make it more readable. The “real” notation introduced afterwards uses more abbreviations to shorten the documentation overhead.

Speedometer

… (some requirements on the speedometer)

<Variation Point: Gulf State Speed Warning>

Criteria: Country

Alternatives:

a. Gulf States → If speed exceeds 120 km/h, there should be a warning tone, which is withdrawn after speed falls below 115 km/h.
b. Rest of World → None.

Selected alternative: Open (depends on sales country)

Dependent variation points: None

<End of variation Point>

… (some more requirements on the speedometer)

Figure 2-19: Extract of a technical specification with variation point.

As we concentrate on specifications, not all binding times mentioned above make sense here. Instead we have to think about more specific binding times. In our context we can identify the following two binding times:

- *Specification time*, i.e. the decision has to be made along the specification process. By means of these decisions, product management defines the new (prime) product.
- *Order time*, i.e. the decision is made when the product is ordered. The buyer’s decision defines the specific product variants that have to be built in the final product.

By postponing decisions to order time, we define a whole set of products (i.e. some product variants) instead of a single product. Here it is irrelevant whether the product is built after the customer has placed his order (e.g. a vehicle, which can be customised with additional equipment) or the customer selects his specific product amongst a set of products (e.g. a standard and a high-end version of the same DVD player).

Now we may ask the question whether it makes sense to make variation points explicit that have to be bound at specification time. If the scope of a specification is a prime product with its product variations, there is no need to make these variation points explicit. If we consider a product line, variations points that are bound at specification time are the means to define the product line itself.

Figure 2-20 illustrates decision making along the product creation process graphically.

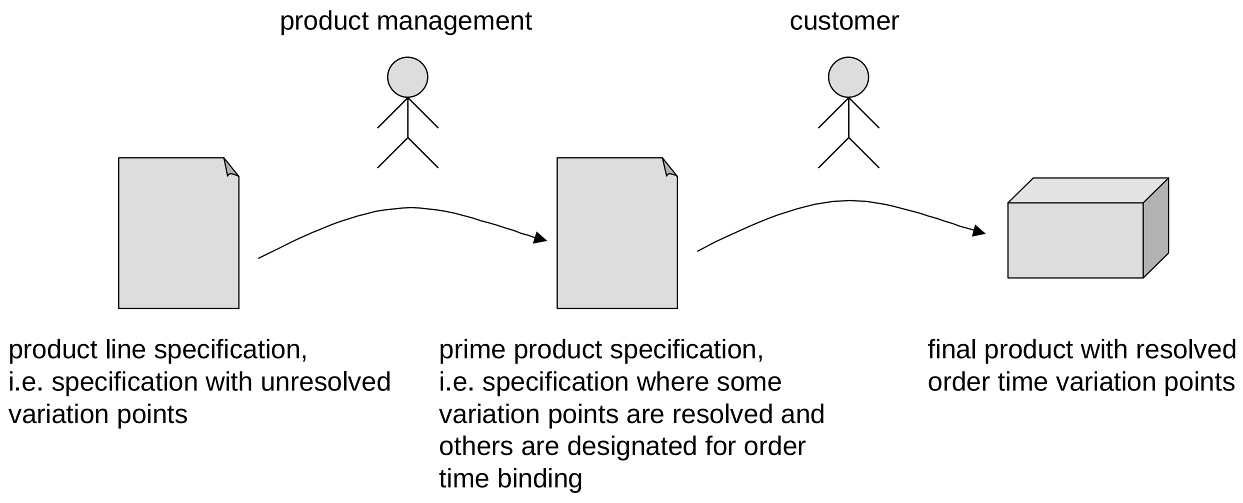

product management
customer
product line specification,
i.e. specification with unresolved
variation points
prime product specification,
i.e. specification where some
variation points are resolved and
others are designated for order
time binding
final product with resolved
order time variation points

Figure 2-20: Product refinement process.

Variation Point Notation

As mentioned already, we propose to embed variation points in the specification by means of some formalised notation. To ease the use of explicit variation points, the notation should be both brief and comprehensible. The general pattern of a variation point is

[Variation Point: <Name> **(**<Criteria>**) ::** <Dependent variation points>
<List of alternatives with its corresponding requirements as pairs a→b, separated by semicolon; selected alternatives are marked with an asterisk or the name of the prime product or product variant> **]**

Creating Variation Points

We propose the following procedure. The (requirements) engineer gets a change request and proceeds as described above. But during this change process the engineer asks whether the change request corrects a mistake in the specification or introduces a variation meaning that the original part could still be relevant for future or other products or variations. If the engineer is convinced that the change describes a variation, the new specification will not replace the existing ones, but a variation point will be introduced.

A variation point may also be introduced when new aspects of a system have to be developed (see Section 2.3.5.2). The description of the new aspect is linked to the variation point, at which the existing specification needs to be changed.

Having the layered approach introduced in Section 2.3.2 in mind, we need related variation points. This means, that a variation point at level X implies one or more variation points at level X+1. If, for example, there is a variant feature (i.e. variation point that offers the selection amongst several features) we also need related variation points in the Use Case section and in the system requirements section that refer to the feature level variation point.

Types of Variation Points

Not all variation points are the same. As we have already seen, we can distinguish variation points with respect to the binding time associated to them. Variation points decided about at specification time may be called *product line variation points*. Variation points that are resolved at order time are *product variant variation points*.

Over time it is likely that the set of variation points in a specification increases. As a consequence, the set of alternative products emerges, too. Managing and using such specifications becomes more difficult. Thus it would be desirable to reduce the set of variation points. A means to do so provides the classification of variation points according to their strategic relevance. We can distinguish strategic and historic variation points. A *strategic variation point* is a variation point that expresses the chance to select product properties within the product definition process. A *historic variation point* on the contrary expresses a variation that differentiates former product properties from recent ones. The reason for introducing such variation points are interleaved product development processes.

Example: *Currently we are specifying a hair dryer FAN-1 with one fan speed. As the development is on the way, strategic product definition decides that our new hair dryers will be equipped with two fan speeds (low, high). Thus we incorporate a variation point in the product specification, which allows us to select between one and two fan speeds. However, the variation point is only needed because FAN-1 specifications are still required. If FAN-1 development has been completed before, there would have been non need to introduce a variation point. Instead the change from one to two fan speeds would have been made directly.*

Whereas strategic variation points give us the means to handle a set of products (and their specifications) in a systematic way, we can remove historic variation points when the development and maintenance of "historic" products has been completed. It can also likely be that strategic variation points become historic variation points over time.

The management of these different variation points is still a matter of research, addressing questions like: How can the developer be supported in distinguishing between these two kinds of variation points? How can the evolution of a strategic variation point to a historic variation point be comprehended? How can the interdependencies between different variation points be kept current? …

2.3.6 Case Study on Evolving Specifications

To accomplish a change request constantly, it is essential to keep an eye on the rules, which have to be heeded and which are specified in 1.1.2.4 and the following.

The entering point for making changes in a given requirement-tree is of course at the goal level.

2.3.6.1 Case 1: New Competitor Feature

Change request: "Visualise activated adaptive wiping at car start time by a single wipe move" to the wish-wiper example given in Section 2.3.3. The idea behind this change request is that a driver who starts driving shall not be scared by a surprisingly starting windscreen wiper.

To embody this change request in the given specification, we have to traverse the existing specification in a top-down manner like our abstraction layer model. First we consult the vision and scope layer. The question in this layer is: "Why do we want this change request and what is its goal?". The given change request does not add another goal – instead it confirms goal G-1.2 (minimise driver distraction). By activating the wiper once at the beginning of a car ride, the driver becomes aware of the fact that adaptive wiping is activated. Now the driver can either deactivate this function or he is at least aware of its operation and will not be distracted if wiping starts autonomously later on. The scopes are not affected by this change request.

Then we have a look at the feature list. Here we see that the change request introduces a new feature, namely F-015 (Adaptive Wiping Awareness) which refers to G-1.2. A conflict of aims is not noticeable between the new feature and the old features so it is not necessary to compare the “conflict system requirements” later with the newly created system requirements. Simultaneously we look at the Use Cases and the effects that appear here. This change request also introduces a new Use Case, too, namely

UC-5a The driver sits in the car

UC-6a The driver turns the key in position “ignition on”

UC-7a The adaptive wiping is not selected such that nothing happens

UC-8a The driver starts the engine and drives away

UC-9a The driver arrives at his destination

UC-10a The driver stops the car and turns the key to position “ignition off”

UC-11a The driver gets out of his car and locks the car. END OF USE CASE.

UC-7b The adaptive wiping is selected. Immediately when the driver turns the key in position “ignition on” a single windscreen wiper move is carried out.

UC-8b Now the driver knows that the adaptive wiping modus is activated and he can turn it off if he wants to. GO ON WITH STEP UC-8a.

After the F/UC layer is completely perfect, we have to accommodate the following layer, named System Requirements Layer. Because there was no conflict of aims between the features by adding the new feature (assuming that in the Use Cases and in the system requirements there are also no conflicts) we only have to add new system requirements which are deduced from the new feature. For this reason the old link structure hasn´t got to be changed but has to be enhanced by a new sub-tree.

(In case of an existing conflict we have to compare all of the “conflict system requirements” with all our new system requirements and dissolve these conflicts.)

2.3.6.2 Case 2: Changed (System-) Requirements and Impact Analysis

The following change request is a more complex example and its consequences are combined with far-reaching structural change:

”A new technical element shall be involved. The windscreen wiper shall be usable by only one thumb wheel. This thumb wheel has on the one limit stop the position off. By turning the wheel, a continuously interval starts between 1 wiping per minute and enduring wiping in the “normal” wipe speed modus, behind the enduring normal wipe speed modus, there is a little turning resistance. After turning the thumb wheel over this turning resistance, the windscreen wiper wipes continuously in “fast” speed mode and after the further turning resistance (to the other limit stop), the windscreen wiper wipes continuously in the “very fast” mode. Furthermore, the wiping speed shall automatically turn up when the driver drives faster (The thumb wheel stays in its position at that time). Driving slower again (the same speed as by adjusting the turning wheel), the wipe speed modus decreases again to the previously adjusted frequency of wiping the windscreen.

What is the application flow to handle this change request?

a) Again we are starting at the goal layer. For all existing goals, no old goal can be identified that is in conflict of aims with our change request. One new goal can be created:

- G-1.9: Simpleness: The windscreen wiper shall be useable in a intuitive and very easy manner

b) Next layer. For all Features we are searching for conflict of aims. Affected Features to change are:

- F-002: 2 Fixed Wiping Speeds

- F-007: Automatic Wiping Speed Reduction

c) Changed features are:

- F-002: 3 Fixed Wiping Speeds and a continuously adjustable wiping modus in speed modus “normal” that are adjustable by the turning wheel.
- F-007: Automatic Wiping Speed tuning up by driving faster and automatic wiping speed reduction by driving slower again or by new adjusting of the thumb wheel by the driver.

d) No new feature can be identified.

e) Now we are looking to the link structure between the goals and the features. New links from goal G-1.9 (Simpleness) can be drawn to feature F-002, F-008, F-010, F-013. No links have to be deleted.

f) New links from the changed features to the goals: From G-1.6, G-1.9 to F-002 and from G-1.1 to F-007. Links that have to be deleted: From G-1.8 to F-007.

g) Same Layer, Use Cases: For all Use Cases we are looking for a conflict of aims. Affected Use Cases to change are the Use Cases that are using the changed features:

  - UC-17 It is drizzling or raining lightly. Driver clears Windscreen of Drizzle by turning the turning wheel into appropriate position. The windscreen wiper wipes the windscreen intermittently according to the build-up of water on the windscreen. Driver switches off windscreen wiping.

    includes Clear Windscreen of Drizzle

  - UC-165 It is raining very lightly. Driver selects Clear Windscreen of Splash by turning the turning wheel into appropriate position. The car wipes the windscreen once.

    includes Clear Windscreen of Splash

  - UC-279 The windscreen is dirty. Driver selects Clear Windscreen of Dirt by turning the turning wheel into appropriate position. The car pumps water on to the windscreen, and wipes it to remove the pumped water and the dirt.

h) Next layer. For all system requirements that are linked with the changed features we are looking for changes triggered by the change request.

i) On every changing of a system requirement we have to pay attention to the incoming links from the input signals and from the parameters and we have to pay attention to the outgoing links from the output signals.

The consistence of the link structure model has first priority and it has to be changed in accurate manner.

2.3.7 Conclusions and Further Work

As a foundation for a framework of requirements for evolving systems, we presented a concept of abstraction layers for specifying software-based systems. This concept will lay the foundation for describing the specification process, both the first development cycle (when building a new system from scratch) but also iterations when coping with evolving systems.

Basing on our experiences gained when applying this approach we got some evidence that the following advantages are likely to achieve:

- Completeness (are all possibilities considered?)
- No appearance of discrepancy (are there any requirements conflicts?)
- Consistency (do the requirements match?)
- Adequacy (do the requirements match to the desires of the customers?)
- Correctness (do the requirements and the interfaces match to other requirements and subcomponents?)
- Understandability (are the requirements easy to understand?)

- No redundancy (does a modification have consequences to more than one different location?)
- Traceability (are the requirements traceable up to their origin?)
- Maintainability (which requirements have to be modified according to a specific change request?)

To exploit these benefits to their full extent we need a sufficient tool support as well. Especially the handling of the links is a crucial element and has a major impact on the daily handling of this approach.

3 Modelling Requirements

Requirements describe what properties a system under development is expected to have once its development is completed. Several sources of requirements influence development: requirements from customers, requirements that have to be taken into account internally at the system developer, organizational as well as technical constraints, and more. All of them can be divided into several classes. A frequently (also in EMPRESS project) used classification is the classification into functional and non-functional requirements. We will use the following definition of functional and non-functional requirements

A **Functional Requirement (FR)** is a requirement that can be expressed using a precise and executable model. All other requirements are **Non-Functional Requirements (NFRs)**.

Note that this does not mean that only executable models are functional requirements. Examples for functional requirements can also be expressions like "the system must not ..", or "it does not take longer as .. to ..". where the ".." are expressions of the model. More precisely, functional requirements can be classified into **constructive requirements** that can be used to construct a model, and into those that cannot be used to construct a model. We call them **constraining requirements**. Constructive requirements can be further classified, according to the models used and to their kinds (see Section 3.2.3.1). Constraining requirements can be classified according to their quantification: some constraints ("**existential**") can be shown to hold using a single scenario / test, other constraints ("**universal**") can only be verified using a complete test suite. Universal requirements can be falsified using a single counter-example. This classification is depicted in the scheme in Figure 3-1.

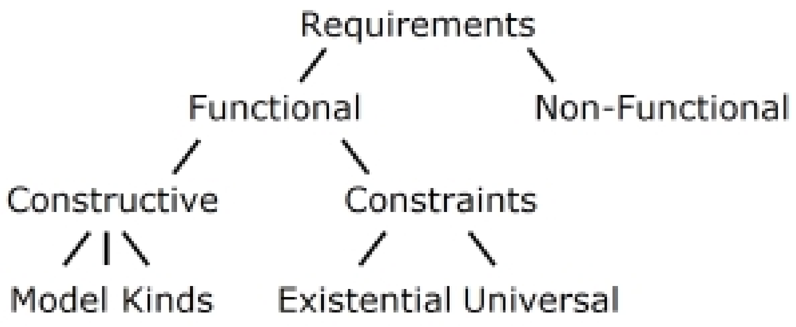


Figure 3-1: Requirements Classification

In Work Package 3.4, Validas developed a tracing mechanism, that allows to evaluate the requirements coverage of constructive requirements, existential and universal requirements on a given test suite for the model. Of course, universal coverage can only be measured on the models with respect to the executed test scenarios (a kind of closed world assumption that is called complete coverage assumption in this context).

3.1 Specifications of Software Requirements: a rigorous approach based on UML

3.1.1 Foreword

UML is being increasingly used to model requirements of software systems. Specifically for this purpose, UML provides the "use case" diagrams [Jac95]. Unfortunately, use cases suffer from several limitations, amply described in the literature [Ber], [Com97], [Kor98], [Jac01], [Gli00]:

- They capture only functional requirements. Actually they often lead the modeller to adopt a functional specification style.
- The correct granularity of the use case is not always clear, so that it is very easy for the modeller to introduce unnecessary details or (more often) to skip important features.
- Use cases are even less formal that the rest of UML, so that misinterpreting them is very easy. This is partly due to the fact that the system description provided by the use cases diagrams is so simplified that usually it must be complemented by plain text. As a consequence, a lot of effort has been devoted to the formalisation of use cases [Hur97].
- Use cases work well when modelling discrete services used in clearly delimited episodes. Unfortunately, in many real cases, problems do not consist of use cases, for instance when it is required that the machine continuously interact with the problem domain [Jac01].
- Use cases suffer from several problems concerning structuring, decomposition, information flow and interaction among use cases, etc. [Gli00].

Another big limit of use cases is that they are not object-oriented. In a UML-based development this creates a gap between the use case-based requirements elicitation phase and the following object-oriented phases, where class and object diagrams are employed. In practice once the use case modelling phase is completed, the user still has to identify and describe classes, attributes, methods, etc.

In practice the adoption of use cases for requirements representation calls for an additional phase that converts the non-object-oriented requirements specifications into a UML model of the same requirements. This phase is clearly non trivial (and therefore calls for some effort), as long as the object-oriented structure of the system has to be identified and documented, and the requirements themselves have to be extracted (and often interpreted, completed, disambiguated) form the use case text.

The limitations described above can be summarised as follows:

1) The representation of requirements is not at all formal nor rigorous. In fact requirements are mainly described by the text that illustrates the “courses of action”.
2) It is not clear how to move from use cases to the object-oriented models required by the following phases of a UML-based development process.
3) Even in case a good object-oriented model is built, the traceability between the use cases and the elements of the object-oriented model is generally not very clear.

These observations suggest that employing use cases in a UML-based development is not very convenient. With respect to the representation of requirements as pure text, use cases only enforce a bit more systematic approach.

On the other hand, several rigorous approaches have been proposed in the last decade []Jac95],[PM95],[GGJ00] that do not suffer from the drawbacks mentioned above. These approaches are nevertheless not very popular (at least, not if compared with UML). The problem with these approaches is that industry generally needs methods, techniques and tools which are readily applicable and become productive in the short term. On the contrary, the rigorous approaches mentioned above, especially when employing formal notations, are considered difficult to learn and/or to use, and therefore are perceived as not immediately productive. Some other methods, like the problem frames proposed by M. Jackson [Jac00] provide a good conceptual approach to requirements description, but are rather weak with respect to the linguistic tools to be used in practice. The lack of a linguistic proposal leaves the user with the choice of a language: in practice some users adopt logic languages like Z, others simply express properties informally, in natural language. Both solutions have drawbacks, in particular they do not make the transition to the specification and design phases very easy, especially when such phases are carried out using UML as a modelling language. Also traceability between problem frames and UML models is often problematic.

Therefore, we explore the possibility of removing obstacles to the adoption of rigorous approaches by making them applicable in the context of the familiar UML language. In other words, we propose to discipline the use of UML by incorporating the principles of rigorous approaches in the construction of UML requirements models. The goal is to increase the quality of requirements models, while keeping UML as the modelling notation.

The foreseen advantages of the suggested approach are the following:

- With respect to use case modelling, requirements are described in a more rigorous and precise way; moreover, the representation is object-oriented from the very beginning, thus enabling a smooth development process, where traceability of requirements is easier.
- With respect to formal methods, the proposed approach is a trade-off between formality and accessibility. Requirements representation is less formal (yet rigorous), while the usage of UML facilitates the adoption of the approach, and makes it easy to keep trace of requirements in the design models.

The approach is illustrated by means of a sample problem. Requirements of the sample system are described in terms of the requirements modelling techniques proposed by Michael Jackson [Jac95]. Then a UML model of the requirements of the sample application is illustrated. This model is made of a set of UML class and state diagrams, as well as statements expressing invariants, operations specifications and constraints, written in OCL (the Object Constraint Language, defined as part of UML [OMG01]).

3.1.2 The sample software system

We consider the patient monitoring system of an intensive care unit (ICU), a well-known reference problem, which was first introduced in [SMC74] and then used in [GGJ00] and in[Jac01].

The requirement is that a warning system notifies a nurse that a patient’s vital signs indicate a critical situation. To do this, there is a computer-based system equipped with sensors that are able to capture the patient's vital signs and an actuator capable of sounding a buzzer. The system can be programmed to sound the buzzer on the basis of data received from the sensors. The system has also to record the data received from sensors, so that a physician can later see the evolution of the patient’s situation over a period of time.

There is also some knowledge of the world:

- ❑ There is always a nurse close enough to the nurse's station to hear a buzzer sounded there;
- ❑ It is known how the vital signs from the patient must be interpreted in order to determine when the intervention of the nurse is needed. For simplicity, only the patient’s heart rate is taken into account, the situation being considered critical when the heart rate is less than a given minimum value. Extending the problem description in order to take into account other factors (such as blood pressure, temperature, etc.) is relatively easy.

3.1.3 A rigorous approach to requirements modelling

In this section, the requirements informally given in the previous section are modelled according to the approach proposed in [Jac95] and then formalised in [GGJ00]. Actually, most of the contents of this section are taken almost literally from [GGJ00] and [Jac95], in order to make the article as self-contained as possible.

The elements of the system can be represented by means of a context diagram, as shown in Figure 3-2. Several alternative context diagrams are possible; however the proposed diagram (a slightly modified version of the diagram reported in [Jac01]) models well the physical structure of the system.

According to Jackson's notation, boxes represent domains. The domain with a double vertical stripe is the machine, while the domain with a single vertical stripe is a designed domain, i.e., a domain that is subject to design decisions of the developers.

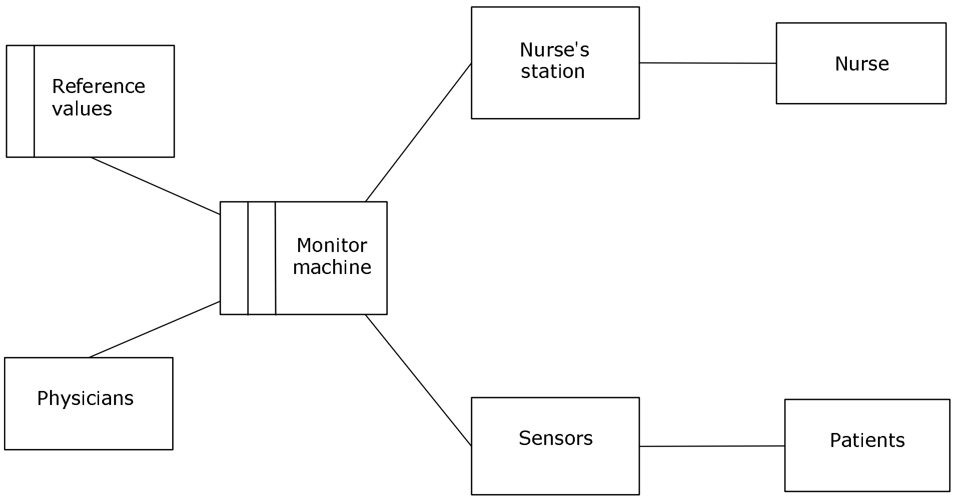


Figure 3-2. Context diagram for the sample system.

The context diagram includes items which belong to the environment as well as to the software system to be developed. The environment and the system are characterised by five artefact types [GGJ00]:

- ❑ Domain knowledge (W) provides presumed environment facts;
- ❑ Requirements (R) indicate what the customer needs from the system, described in terms of its effect on the environment;
- ❑ Specifications (S) provide enough information for a programmer to build a system that satisfies the requirements;
- ❑ A program (P) implements the specification using the programming platform; and
- ❑ A programming platform (M) provides the basis for programming a system that satisfies the requirements and specifications.

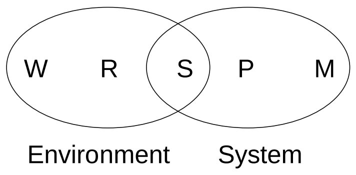


Figure 3-3. Five Software artefacts.

The five types of software artefacts are arranged as follows (see Figure 3-3):

- The environment is characterized by the Domain Knowledge, Requirements and Specifications;
- The system is characterized by the Specifications, Program and Machine;
- The Specifications act as the interface between the environment and the system.

In order to describe precisely the scope of requirements and specifications, the environment and the system can be further decomposed into phenomena *e* belonging to the environment and phenomena *s* belonging to the system. Phenomena e_v are visible to the system, while all the other phenomena in *e*, e_h, are hidden from the system. In a similar way the *s* phenomena are decomposed into s_v and s_h [GGJ01] (see Figure 3-4).

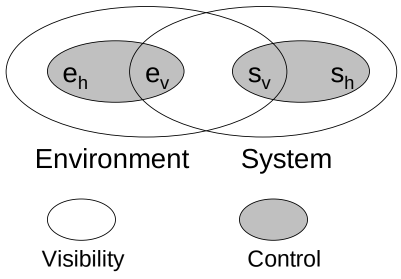


Figure 3-4. Visibility and control for designated terms.

From the above classification stems that Requirements care expressed in terms of e_h, e_v and s_v, while Specifications are expressed in terms of e_v and s_v only.

The designated items reported in Figure 3-2 can thus be classified as follows:

- ❑ e_h : the nurse, the physician and the patient.
- ❑ e_v : patient's vital signs detected by sensors, the set of reference values for vital signs.
- ❑ s_v : the buzzer at the nurse's station, the recording of data from sensors.
- ❑ s_h : internal representation of data from the sensors.

This classification is based on the controllability and visibility of items. Thus, the sensor is in e_v because it is controlled by the environment (namely the patient) and is visible by the system. Similarly, the data log and the buzzer are belonging to s_v because they are controlled by the system and visible to the environment.

Requirements are expressed in terms of e_h, e_v, and s_v. In fact, the requirements of the sample system –expressed in an informal way– say that:

- ❑ The nurse must be warned whenever the patient's situation falls out of the safety conditions established by the reference values.
- ❑ The patient's situation, as expressed by means of a set of vital signs, must be recorded.
- ❑ There is always a nurse close enough to the nurse's station to hear a buzzer sounded there; the sound of the buzzer is interpreted as a warning concerning the patient's situation.
- ❑ The patient's vital signs are measured by a set of sensors.

Specifications are expressed in terms of phenomena which belong both to the environment and to the system, i.e., e_v, and s_v. For the sample system:

- ❑ If the data provided by the sensor falls below the thresholds indicated by the reference values, then the system must sound the buzzer.
- ❑ Data provided by the sensors must be recorded, and then must be available for browsing.

Requirements and specifications of the case study were specified informally at the end of the previous section. In order to make the approach to the definition of requirements and specifications more rigorous, it is necessary to model clearly and precisely the elements of the model that are mentioned in requirements and specifications.

For instance, domains are characterized by:

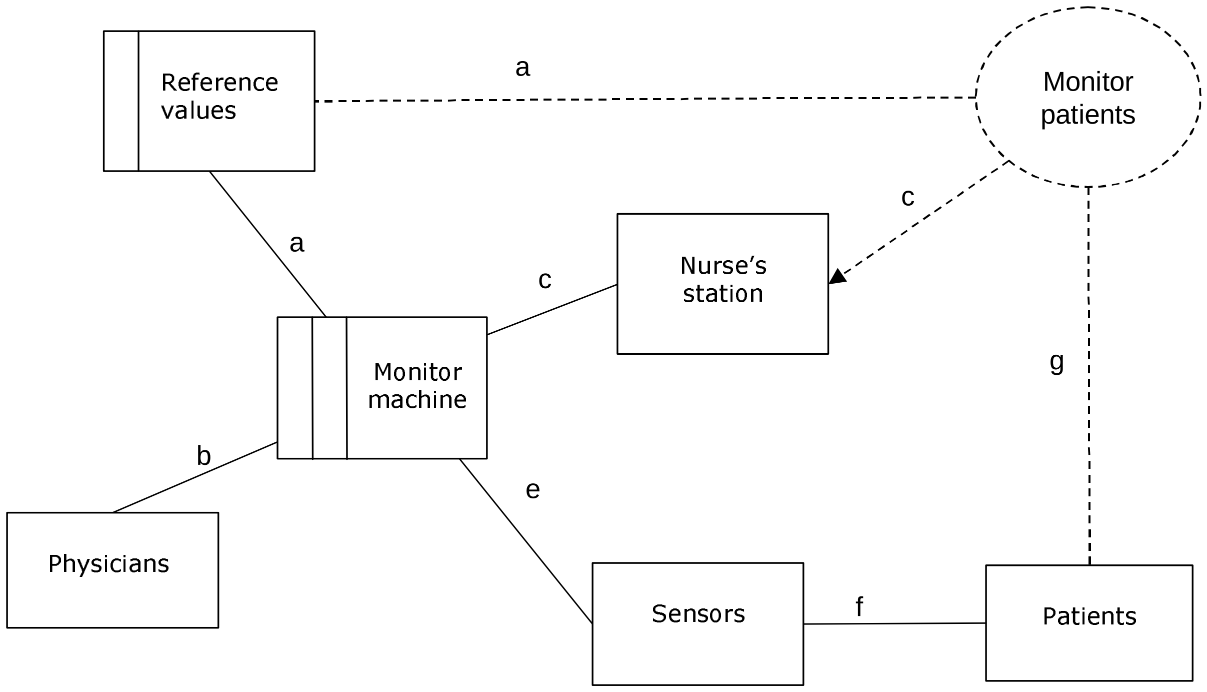
- entities (e.g., the nurse's station, the patient, …);
- attributes (e.g., the patient's heart rate);
- relationships (e.g., the fact that a given sensor is attached to a given patient);
- events (e.g., the buzzer is turned on);

- rules or causal laws (e.g., the patient can be monitored only after the period and ranges have been defined).

The structure of the system is mainly determined by the phenomena (events, states, values, …) shared between domains.

Of course the final goal is to express requirements. For this purpose problem diagrams are introduced: they extend context diagrams and provide a first analysis of the problem describing requirements, and showing the connections between requirements and domains.

Requirements involve shared phenomena, thus a notation is introduced to identify phenomena.



a: Period, Range, Patient name
b: EnterPeriod, EnterRange, EnterPatientName
c: Notify
d: RegisterValue
e: FactorEvidence
g: VitalFactor, Patient

Figure 3-5. Partial problem diagram for the patient monitoring system.

Figure 3-5 illustrates a problem diagram describing part of the patient monitoring system. It is possible to note that the shared phenomena (e.g. Period, Range, Notify, …) are explicitly modelled.

3.1.4 Modelling requirements with UML

The problem with the approach described in the previous section (i.e., Jackson's) is that:

- The "internals" of domains are not satisfactorily modelled. There are no specific notations to distinguish entities, attributes, operations, events, etc.
- Elements of domains, shared phenomena, etc. are identified only if they are involved in the with reference to requirements. The modeller is not induced to explicitly model a characteristic of a domain, unless it is not involved in a requirement. This may result in a poor understanding and documentation of the problem domain.
- Even in the more detailed diagrams (like the one reported in Figure 3-5) the nature of the phenomena mentioned in the requirements could be not perfectly clear (e.g., attributes and operations are mixed together).

This section discusses the hypothesis that UML can be employed to model the domains and phenomena in order to support a very expressive though rigorous description of requirements. Note that the principles of Jackson's approach are maintained: only the underlying notation is changed.

3.1.4.1 Structure of the system

The designations reported in the context diagram of Figure 3-2 can be naturally translated into UML classes. Figure 3-6 illustrates a first draft of the class diagrams corresponding to the context diagram of Figure 3-2. Note that:

- The names of some classes do not correspond exactly to those of the domains appearing in Figure 3-2 and Figure 3-5. For instance, the class Buzzer replaced the Nurse's station in order to remind that the machine does actually control (only) the buzzer.
- The relationship between the physician and the reference values was made explicit. This is reasonable, since the physician is the originator of the reference values, in the same way as he/she is the user of the records contained in the patient's log.
- In Figure 3-6 all the shared phenomena are represented by classes. In general some phenomena could be better modelled by other elements of a UML model (like interfaces, attributes, etc.). However in the initial model it is often simpler–though safe–to model phenomena as classes. This mapping has the advantage to clearly classify classes in the e_h, e_v, s_v sets. Later on the modeller is free to replace classes with other elements of UML.

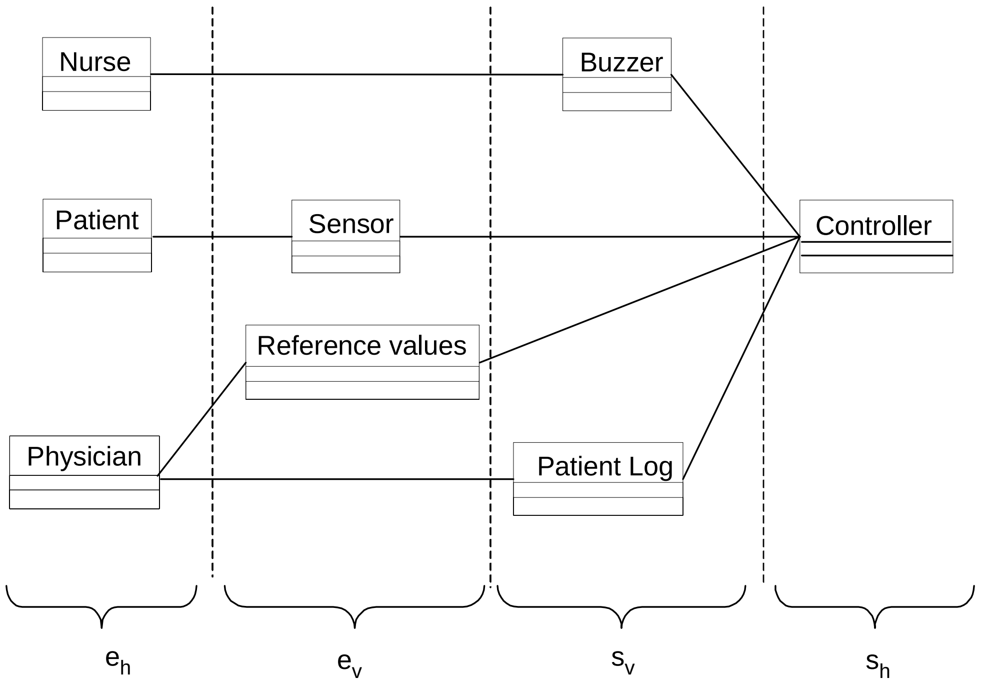


Figure 3-6. UML class diagram.

A note on the methodology

The diagram reported in Figure 3-6 was derived from the context diagram given in Figure 3-2. In general it won't be necessary to build a context diagram before proceeding to the construction of the UML model. In particular, it is convenient to model the context diagram directly using UML constructs. For this purpose it is possible to proceed as suggested in [Jac01]

Proceeding with the construction of the UML model, the next step consists in enhancing the class definitions in order to include attributes and operations which naturally belong to the original designations. Attributes are necessary to represent domain states and values which are mentioned in requirements. Operations represent events and transformations which are also mentioned in requirements. Both attributes and operations may concern e_h, e_v and s_v, but not s_h. The enhanced class diagram is reported in Figure 3-7.

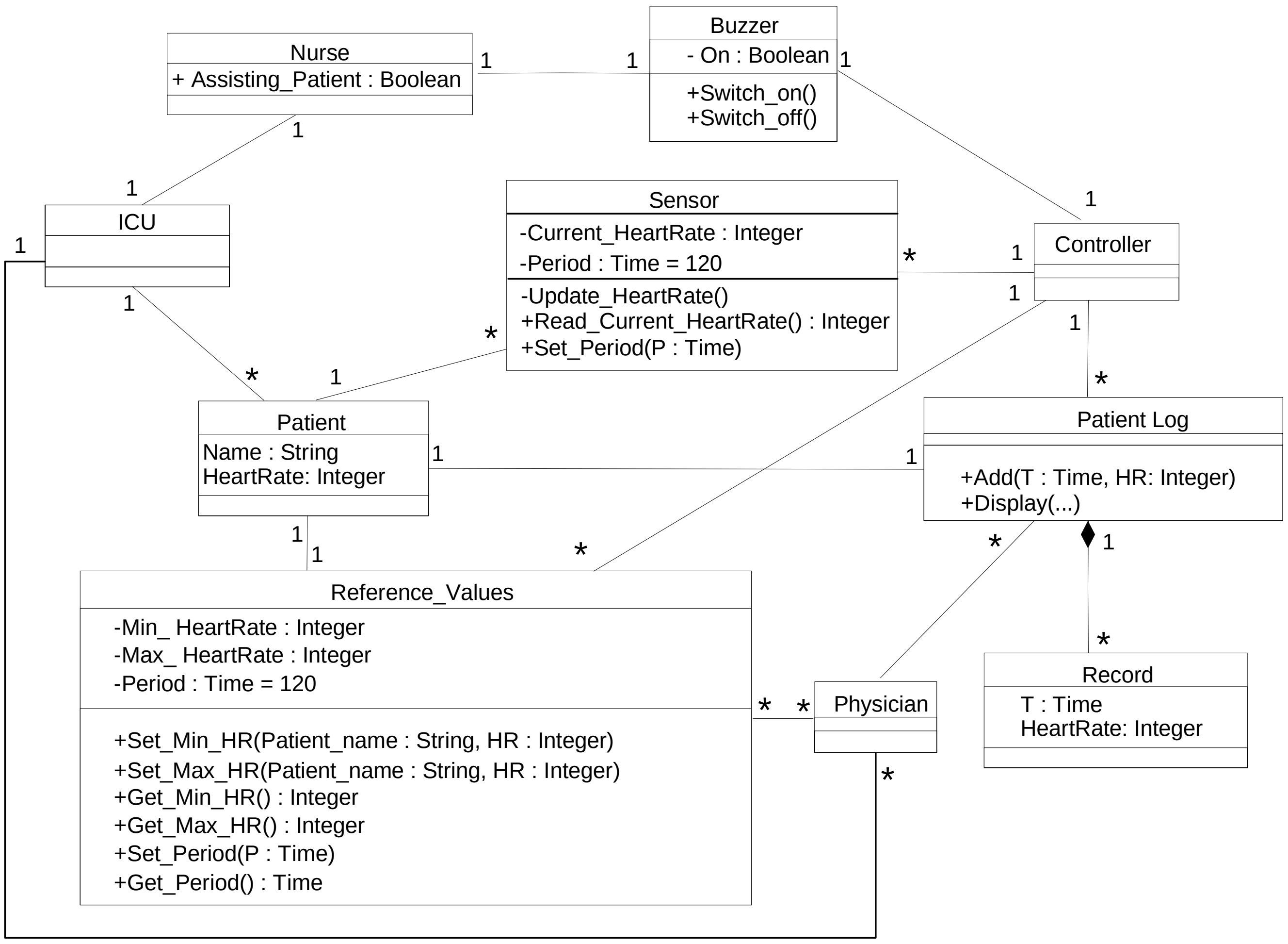


Figure 3-7. Complete UML class diagram.

It is interesting to note that the construction of the diagram requires a good understanding of the problem domain. In other words, the need to build a detailed and precise representation of the system induces the modeller to improve his/her insight in the problem domain. For instance, the need to express the cardinality of the association between the `Reference_Values` class and the `Controller` class causes the modeller to investigate whether a unique set of reference values is valid for all the patients or if every patient has his/her own values (the latter hypothesis is being assumed in our model).

The meaning of the UML class diagram reported in Figure 3-7 is quite intuitive and does not require much explanation, except for the following issues:

- Class `Record` was introduced, although no record domain was present in the original context diagram. Class `Record` was introduced to describe in greater detail the nature of the `Patient_log`. In this case we simply decided that the class diagram needs to be more detailed than the original context diagram. If we do not introduce class `Record` in the diagram at this stage we will need to do it later, e.g., when we specify the behaviour of the system when data from the sensor are read.
- Class `ICU` (Intensive Care Unit) was introduced in order to explicitly represent the environment, which is an implicit concept in the context diagram. In this way we can specify–by means of the cardinality of the association with `Patient`–that several patients can be present in the system.

- The visibility of properties is specified. Information hiding does not make much sense in a *problem* description (see the discussion in [Jac01]). Nevertheless, the specification of public operations (and private attributes) makes it easier to indicate explicitly which operations are possible on a set of related data belonging to an object. This is useful for several purposes: an abstract description of the class is made possible (i.e., the class is described as an abstract data type); the meaning of operations can be specified precisely by means of pre-conditions and post-conditions (written in OCL); it is easy to specify who invokes an operation and who does not: for instance, we may use a state diagram (Figure 3-11) to state that the physician can modify reference values, while others cannot.
- The visibility of properties is specified. Information hiding does not make much sense in a *problem* description (see the discussion in [Jac01]). Nevertheless, the specification of this kind of information makes it possible to describe classes as abstract data types. Moreover, it is easy to specify who invokes an operation and who does not: a public operation indicates that an external object will perform the operation, while state diagrams specify which classes can issue the operation request. For instance, we know that establishing the minimum safe heart rate is a public operation (`Set_Min_HR`, Figure 3-7): we can then specify precisely by means of pre-conditions and post-conditions (written in OCL) the meaning of the operation, and use a state diagram (Figure 3-11) to state that the physician can modify reference values. Other objects cannot modify reference values, since their state diagrams do not include the invocation of `Set_Min_HR`.
- The arguments of the Display operations that can be performed on registered data (`Patient_Log`) have been omitted for simplicity. In a real case there could be several display operations available.
- Real systems are often too complex to be modelled by a single class diagram. In such cases we would exploit the subsystem and package constructs provided by UML in order to suitably decompose and structure the model.

The diagram specifying the structure of the system can often be complemented by some OCL statements that clarify the nature of classes and/or associations. For instance, in our case it is convenient to establish an invariant on class `Patient_Log` imposing that all the records referring to the same patient have different times (i.e., records are not replicated):

```
Context Patient
  inv: associated_Log.Record->forAll(r1,r2 | r1<>r2 implies r1.Time<>r2.Time)
```

It must be noted that while the diagram reported in Figure 3-7 is well suited for expressing requirements, some subtle modifications have to be introduced to make it suitable for expressing specifications. Consider for instance the case when multiple patients are present and multiple sensors are attached to each patient: the (unique) controller has to distinguish which subset of sensors refers to which patient, in order to calculate whether a warning is needed, and to associate the data to be recorder with the “owner”. This means that while the patient is actually in e_h, his/her identity is in e_v. This is reasonable, since we have an association connecting an element in e_v with an element in e_h. When removing the element in e_h, in conformance with the definition of Specifications, we also remove the associations, but one end of the association belongs to e_v, thus it cannot be always safely removed. The problem can be easily solved by preserving the knowledge contained in the association end: in our case introducing in the Sensor class the identity of the associated patient. Of course, the name of the patient would be the best choice, since the physician probably wants to use the name of the patient as a research key to find the related data in the patients’ log.

3.1.4.2 Behaviour of the system

Once the structure of the system and the elements of the system have been described by means of a set of class diagrams, requirements can be specified by defining the behaviour of the classes that appear in the state diagrams. The behaviour of a class can be specified by means of UML state diagrams, and by imposing constraints on objects’ values and states by means of OCL.

As far as our case study is concerned, here we concentrate on the requirements for the behaviour of the running system. The initialisation phase is trivial (it just involves specifying that the buzzer is initially silent, the log empty, and the reference default values properly set), therefore it is not treated explicitly.

3.1.4.2.1 The buzzer and the nurse

The state diagrams for classes `Buzzer` and `Nurse` are reported in Figure 3-8.

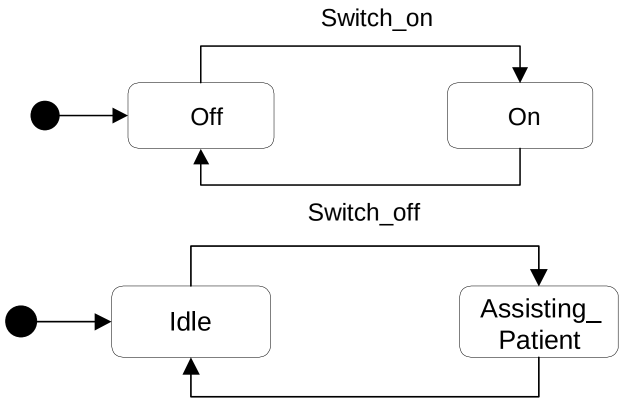


Figure 3-8. State diagrams for class Buzzer (left) and Nurse (right).

The behaviour of the nurse as a consequence of the state of the buzzer is specified by the following OCL statement:

```
context Buzzer
 inv: (oclInstate(Off) implies Nurse.oclInstate(Idle)) and
      (oclInstate(On) implies Nurse.oclInstate(Assisting_patient))
```

This statement (together with the state diagrams) indicates the combinations of states of the nurse and buzzer which are possible in the real world. The “abstract” knowledge provided by the OCL statement is enough to let us transform user requirements (which involve the nurse) into software specifications (which do not).

Note that in order to make the OCL code easier to read, a little syntactic simplification is adopted: instead of writing

```
context Nurse
Self.associated_buzzer.associated_controller.associated_sensor.associated_patient
...
```

which is rather awkward to read, we write simply

```
context Nurse
  Patient
  ...
```

as long as it is not ambiguous. This simplification is applied in the rest of the paper.

Note that the OCL invariant reported above is necessary, since the state diagram of the nurse does not indicate when and how the state changes. This is perfectly consistent with the fact that the nurse is in the part of the problem domain which is hidden from the system. We do not need to have knowledge of the domain so deep as to know the mechanism through which the state of the buzzer affects the state of the nurse. All we need to know (and to model) is that whenever the buzzer is on the nurse is assisting the patients, while he/she is idle when the buzzer is off. In fact the diagram would not change if the buzzer would be located in the office of a supervisor who warns the nurse on duty when the buzzer is switched on: the supervisor can be omitted from the model whenever he/she can be regarded as a reliable communication channel.

We can also note that there are some aspects of the real world which can be "abstracted away" from our model without affecting its validity. For instance, suppose that the nurse–once warned–may locally silence the buzzer (i.e., it does not produce sound, while from the point of view of the controller it is "on"). This is not relevant for the software controller, so it can be ignored. In fact, the specifications of the system will be exactly the same independently from this feature of the buzzer.

Of course it is often possible to embed in the state diagrams a more complex knowledge of the environment, making the OCL code simpler or even not necessary. In our case it is possible to model that the nurse changes state when `warning` or `OK` events are received (see Figure 3-9). As long as `warning` or `OK` events are generated only by instances of `Buzzer` and accepted only by instances of `Nurse`, the diagrams in Figure 3-9 specify exactly the same behaviour as the above OCL statement.

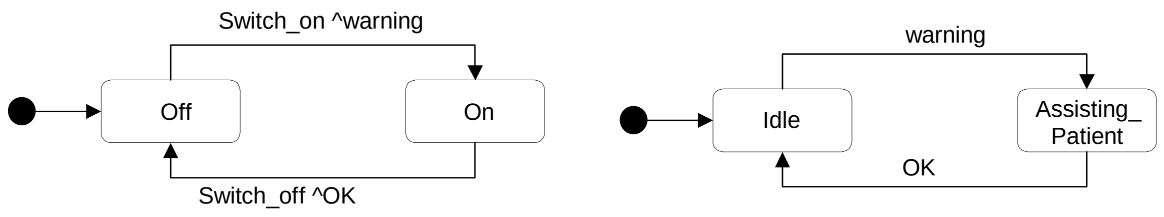


Figure 3-9. State diagrams for class Buzzer (left) and Nurse (right).

3.1.4.2.2 Specification of the main requirement

The main user requirement states that *when vital factors of a patient are out of the range indicated by the corresponding reference values the buzzer is on, otherwise it is off*. Probably the user would most likely say that when the vital factors are out of range the nurse is warned, but we have already established equivalence between the nurse states and the buzzer states, thus we can refer directly to the buzzer. The requirement is expressed by the following OCL statement:

```
context ICU
   inv: (((Patients->exists((HeartRate < Reference_Values.Get_Min_HR() or
              HeartRate > Reference_Values.Get_Max_HR()) and Buzzer.oclInstate(On))
         or
          (Patients->ForAll(HeartRate >= Reference_Values.Get_Min_HR() and
              HeartRate <= Reference_Values.Get_Max_HR()) and
Buzzer.oclInstate(Off)))
```

The following OCL statement specifies that the buzzer is On when the data captured by the sensor falls out of the safety conditions established by the reference values.

```
context ICU
  inv: (((Patients->exists((Sensor.Current_HeartRate <
Reference_Values.Get_Min_HR() or
             Sensor.Current_HeartRate > Reference_Values.Get_Max_HR()) and
          Buzzer.oclInstate(On))
        or
         (Patients->ForAll(Sensor.Current_HeartRate >=
Reference_Values.Get_Min_HR() and
              Sensor.Current_HeartRate <= Reference_Values.Get_Max_HR()) and
          Buzzer.oclInstate(Off)))
```

It is quite clear (and can be easily demonstrated) that the OCL statement above is equivalent to the user requirement, provided that the heart rate value indicated by the sensor

(`Current_HeartRate`) is always equivalent to the patient's heart rate. The relation between these two values is domain knowledge, as it involves only elements of the ICU environment, and can be expressed again by means of OCL statements.

The following statement indicates that the values exported by the sensor are always equal to the values of the patient's vital signs, with no delay.

```
context Sensor
  inv: Current_HeartRate = Patient.HeartRate
```

According to such a statement the user requirement would be satisfied.

3.1.4.2.3 Modelling delays

Of course, it is more realistic that the sensor works cyclically with a given period. For instance, data from the patient are sensed and reported every 3 minutes. In this way, the data used by the system is never older than 3 minutes. Actually, it is reasonable that the physician indicates how often the patient's vital signs should be measured [Jac01].

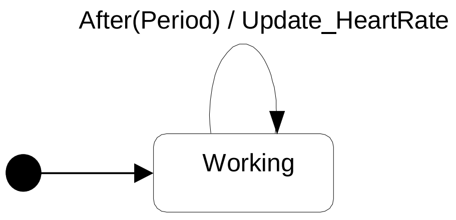


Figure 3-10. State diagram for class sensor.

The behaviour of the sensor and the correspondence of its data to the patient's vital signs are expressed by the state diagram of class Sensor (see Figure 3-10). This diagram indicates that the sensor is continuously updating the value of its internal attribute `Current_HeartRate` with the value of the patient's heart rate. The exact meaning of the operation `Update_HeartRate` is expressed as usual by means of an OCL statement:

```
context Sensor::Update_HeartRate()
   pre:  True
   post: self.Current_HeartRate = Patient.HeartRate
```

Every `Period` the sensor updates the attribute `Current_HeartRate`, which probably corresponds to a register which can be read by the controller. In order to express the fact that the publication of the heart rate value occurs every `Period` we have exploited the After construct: a transition from state A labelled "After(X)" occurs exactly after X time units after the entrance in state A. In more complex cases it could be necessary to introduce a "timer", which sends signals at a programmable rate; timers can be modelled exploiting the mechanism of stereotypes.

The state diagram of class Sensor, together with the OCL specification of `Update_HeartRate` assure that the following user requirement is satisfied: *The vital factors of a patient are measured at a given rate decided by the physician (the default being 2 minutes). Is the measure is out of the range indicated by the physician the buzzer is switched on, otherwise it is off. When the buzzer is on, it remains on until the measure of the vital signs returns in the safe range.*

3.1.4.2.4 Specification of the behaviour of the physician

To be precise, in order to satisfy the above requirement, we still have to specify that the measurement period is actually equal to the corresponding value indicated by the physician. The same applies to the reference values. These facts can be expressed by means of the state diagram of the physician, and of suitable OCL statements, as follows.

The state diagram of the physician is reported in Figure 3-11. The role of this diagram is to specify what operations can be performed by the physician. As long as the physician can *always* set the measurement period or browse the collected data, there is only one state, in which the physician can perform all the operations. Moreover, there is no message or event that triggers the physician's operations (i.e., the physician spontaneously performs the operations), no event has been attached to the transitions in the state diagram. This is not perfectly consistent with UML, which reserves unlabelled arcs for "automatic" transitions. In order to avoid such (small) problem, we could label transitions with "After(random)", in order to indicate that there is no constraint on how long is the time interval between any two actions of the physician.

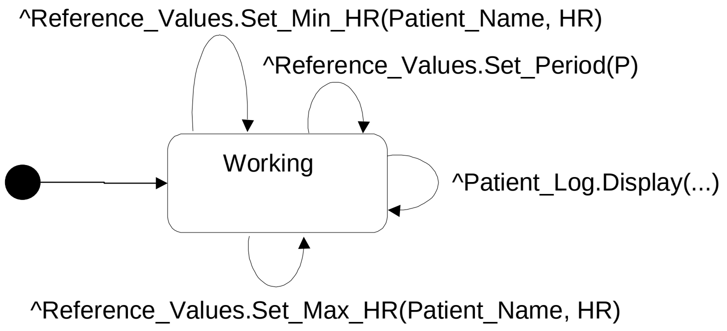

^Reference_Values.Set_Min_HR(Patient_Name, HR)
^Reference_Values.Set_Period(P)
Working
^Patient_Log.Display(...)
^Reference_Values.Set_Max_HR(Patient_Name, HR)

Figure 3-11. State diagram for class physician.

In order to complete the description of the behaviour of the physician, we have to specify the effect of the operations he/she can perform. The following OCL statement specifies the effect of `Set_Min_HR`. The specifications of the other operations are similar or trivial and have therefore been omitted.

```
context Reference_Values::Set_Min_HR(Patient_Name,HR)
  pre: Patient_Name=Self.refers_to.Name
  post: Min_HeartRate = HR
```

Note: `refers_to` is the name of the association connecting `Reference_Values` to `Patient`.

It is interesting to note that logically the physician issues the message invoking `Set_Min_HR` towards all the instances of `Reference_Values`, but only the reference values of the patient mentioned in the message will be affected, because the precondition is satisfied only when the patient associated with the reference values has the name reported in the message. It not specified how the physician knows the name of the patient: this is a part of the domain knowledge that is not modelled.

Finally, we have to indicate that the sensor has to work with the period specified by the physician (and stored in `Reference_Values` as an effect of `Set_Period`). This is done again with an OCL statement.

```
context Sensor
  inv: Period = Reference_Values.Period
```

Note that we specified that the physician does not interact with the sensor. Instead he/she specifies the period, and it is responsibility of the controller to let the sensor know the value indicated by the physician. In other cases the physician could act directly on the sensor: the model should then be changed accordingly.

3.1.4.2.5 Specification of data recording

Up to now we have dealt with the monitoring and warning responsibilities of our system. Now we have to consider the requirement which states that the patient's situation, as expressed by means of a set of vital signs, must be recorded, and should be available for browsing by the physician. According to the analysis reported above, this means that the data provided by the sensor must be recorded and made available to the physician.

The fact that data reported by the sensor are recorded can be specified effectively and simply by means of the state diagram of class sensor, which can be modified as shown in Figure 3-12.

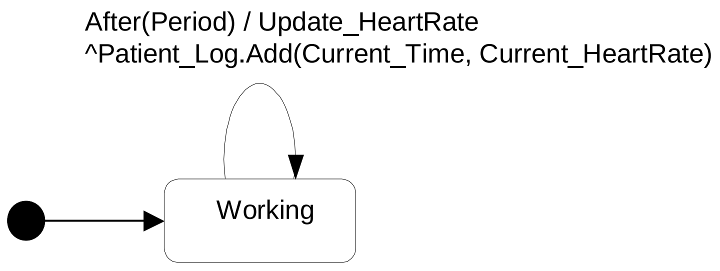

After(Period) / Update_HeartRate
^Patient_Log.Add(Current_Time, Current_HeartRate)
Working

Figure 3-12. New state diagram for class sensor.

The meaning of `Patient_Log::Add` is specified as usual by means of an OCL statement.

```
context Patient_Log::Add(T:Time, HR:Integer)
  pre: self.Records->forAll(R | R.Time < T)
  post: self.Records->forAll(R |
           (self.Records@pre includes R) or (R.Time = T and R.HeartRate = HR))
```

Note that it is not necessary to specify explicitly that the newly created record has to be attached properly to a `Patient_Log` object. This fact is guaranteed by the semantics of the composition association.

3.1.5 Using UML for modelling requirements: a first evaluation

In the previous section we have presented the requirements of the sample system, expressed entirely by means of UML class and state diagrams and OCL statements. The context according to [Jac95] and [GGJ00] was expressed by means of UML class diagrams, which provide a quite satisfactory description of the elements of the problem, their relations and attributes, and even the operations they can be involved in. The behaviour of the system was modelled by means of UML state diagrams and OCL statements.

A first observation is that the resulting requirements actually represent all the relevant characteristics of the problem domain and the software system responsibilities in a clear and non-ambiguous way.

A second observation concerns the combined usage of UML and OCL constructs. While it is expected that OCL plays a fundamental role in requirements modelling, as it is a sort of logic language, it is interesting to note that UML diagrams do not just provide a framework for OCL statements, but contribute actively to the specification of requirements. For instance, the fact that the patient log is readable by the physician derives from information expressed in the state diagram (Figure 3-11) and the specification of operation Display (this specification was not given). Similarly, the fact that a physician cannot add a record to the patient log derives from the absence of transitions in the physician's state diagram sending `Add` messages to the `Patient_Log`. In other words, state diagrams are useful also to state what operations an active element of the problem domain can/will do and what it cannot do.

A third observation is that modelling requirements for the sample system was relatively fast and inexpensive. This is a rather important finding since the cost to specify requirements in a precise or even formal way is generally high.

3.1.5.1 The traditional way: use cases

Let us consider the traditional way of modelling requirements in UML. Two questions arise: how does the proposed approach compare with use cases? Given that other UML diagrams and OCL provide enough modelling power to represent requirements, do we still need use cases?

For answering the first question we need the use case diagram for the patient monitoring system. Such diagram is reported in Figure 3-13. The use case diagram (i.e., the graphical representation alone) is not enough expressive to describe the requirements. For instance, the diagram in Figure 3-13 does not say whether the checking the vital sign measurements against the reference values always results in warning the nurse or not. For this purpose, use cases are usually accompanied by some text describing the “courses of action”. In our case the main course of action will concern the case when the measured values are within limits: they are simply recorded. The alternative course of action will concern the case when the measured values are out of the range established by the reference values: they measured values are recorded, and the nurse is warned.

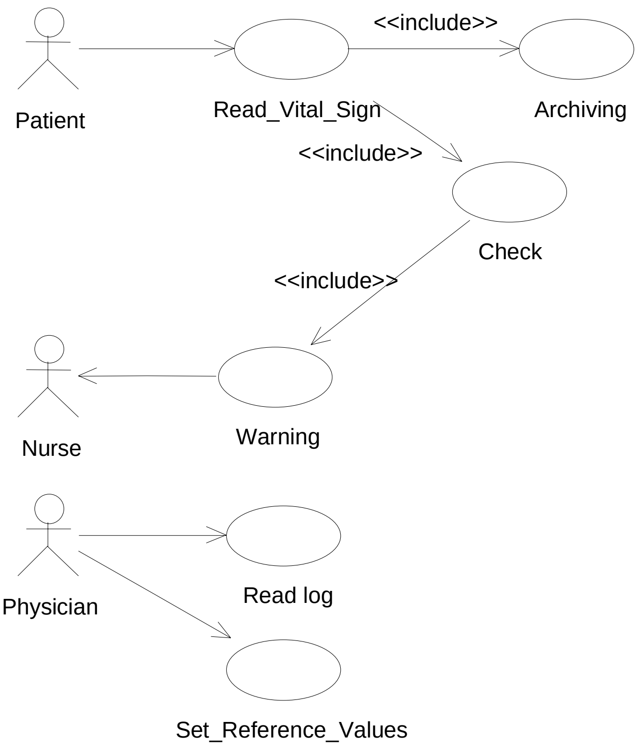


Figure 3-13. Use case diagram for the sample system.

It is quite clear that our model is more precise, detailed and informative.

One could object that use case diagrams are usually complemented by sequence diagrams and collaboration diagrams that improve the precision of the description. This is true, but there are still two big problems:

- Each sequence and collaboration diagram describes one possible behaviour of the system. Therefore requirements are not described in an organic way, rather by means of examples, leaving the possibility that an incomplete set of cases are described.
- Sequence and collaboration diagram are object diagrams: this means that the corresponding classes have to be identified and (at least partially) described *before* diagrams involving instances of such classes can be written. But once you have the described the classes and relations involved in the problem, you could apply the approach proposed here, instead of providing examples that should clarify ambiguous use cases.

Nevertheless, it is not necessary to discard use cases completely. On one hand use cases are too weak to provide a reliable representation of requirements. On the other hand, they can still be used for informal requirements representation, provided that they are then converted into a more precise set of models.

3.1.5.2 User-machine interactions

In our case study we have only simple user-machine interactions. In a more realistic scenario the physician could browse the recorded data, search for patterns or specific conditions, print, ask for average values, etc.

It is possible to specify such interactions by adding the corresponding operations to the `Patient_Log`, and then specify a state diagram for the physician that allows him/her to invoke those operations.

Using class and state diagrams and OCL makes it easy to express precisely properties of the user-machine interaction. For instance the pre-condition that the system can be queried only with respect to monitored patients can be expressed naturally in OCL by means of a pre-condition on the query operation.

3.1.6 Dealing with real-time

3.1.6.1 Problems

Any description of requirements based on UML is not really formal, because UML does not have a well defined semantics. On the contrary, real-time applications often call for very precise specifications, so that they can be formally proved to own the required properties.

Moreover, UML does not currently provide native constructs to express real-time issues. There is a proposal for the “UML Profile for Scheduling, Performance, and Time” [OMG02] [da valutare]. Other proposals [OMG01][SGW99] are oriented to informal and relatively low-level (or implementation-oriented) modelling.

OCL does not provide native constructs to deal with time as well.

3.1.6.2 Solutions: beyond UML and OCL

Currently the easiest way for applying UML in the real-time domain is to define a proper extension of the language. For instance it is possible to replace UML statecharts with well-defined, real-time oriented Timed statecharts [KP92], thus obtaining a new version of the language [BML02a] called UML+, which is suitable for translation into formal languages, which allow the application of model checking and other formal methods [LQV01] [BML02b]. In this way it is possible to achieve both a good definition of the language semantics and a satisfactory representation of real-time issues.

Therefore, for real-time applications, one can use UML+ in the same way as UML was used in the ICU case. For instance Figure 3-14 reports a version of the state diagram for class sensor where constructs of UML+ are used. In fact here we specify that the sensor updates the value of the heart rate with a period variable in the [130, 150] time units interval.

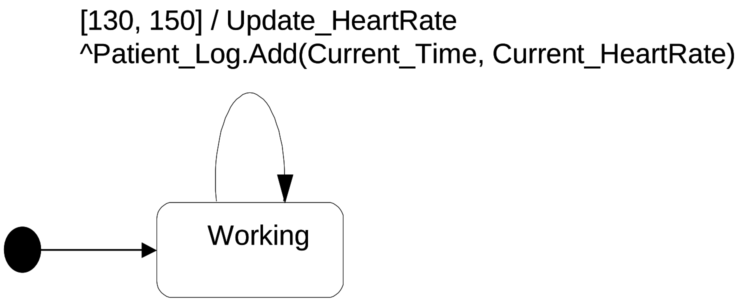

[130, 150] / Update_HeartRate
^Patient_Log.Add(Current_Time, Current_HeartRate)
Working

Figure 3-14. New state diagram for class sensor.

Using UML+ instead of UML is not enough to specify properly real-time systems. In fact, OCL is not sufficient for expressing constraints involving time. Time-related properties can be expressed by means of some powerful temporal logic languages (typically some variant of TCTL [ACD93]).

3.1.6.3 How to use OCL

A first observation concerning OCL is that some properties may require quite complex expressions. A good training is therefore necessary.

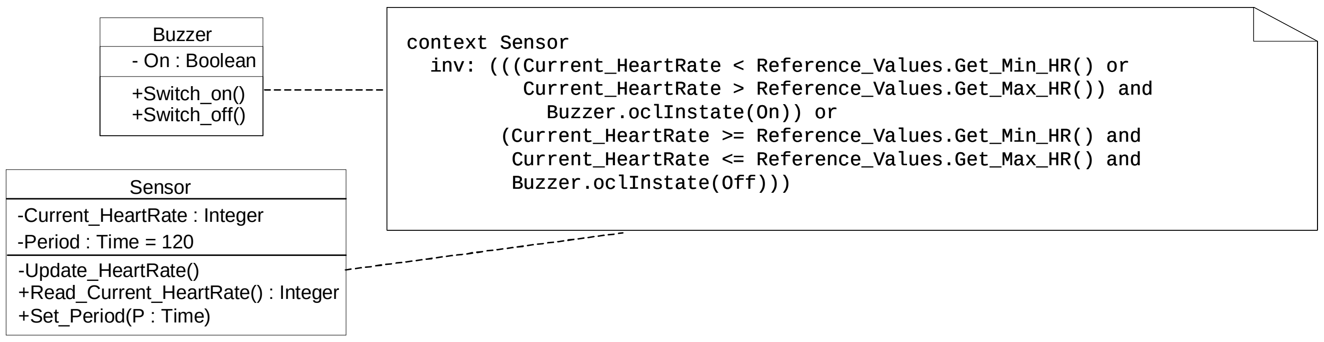
A second observation concerns the syntax of OCL, which allows the user to express the same properties in different ways. In fact, OCL statements are always stated in the context of *one* class, but they often specify properties that involve *several* classes [OMG01]. In other words, the syntax of OCL imposes to attach a property to a class, while the validity of that property is global.

For instance the constraint “alarm is on if and only if measurements are out or range” clearly involves both instances of class `Buzzer` and of class `Sensor`: it can be expressed by means of an invariant on class `Buzzer` or–equally well–by means of an invariant on class `Sensor`, as in the following two OCL statements.

```
context Sensor
  inv:(Reference_Values.out_of_range(Self.Current_HeartRate) and
Buzzer.oclInstate(On)) or
     (not Reference_Values.out_of_range(Self.Current_HeartRate)and
Buzzer.oclInstate(Off))

context Buzzer
  inv: (Self.On and Reference_Values.out_of_range(Sensor.Current_HeartRate) or
       (not Self.On and not
Reference_Values.out_of_range(Sensor.Current_HeartRate))
```

This way of describing properties is effective, but presents a little problem: whenever an OCL statement expressed “in the context of” class A involves elements of classes A and B, you could need to modify it as a result of a modification in class B. This is not very intuitive, but, as long as you know this possibility in advance, it is not terrible. Moreover, in UML you can graphically attach a constraint to both classes (as shown in Figure 3-15), thus reminding the user that the scope of the constraint involves two classes.



```
context Sensor
  inv: (((Current_HeartRate < Reference_Values.Get_Min_HR() or
          Current_HeartRate > Reference_Values.Get_Max_HR()) and
            Buzzer.oclInstate(On)) or
        (Current_HeartRate >= Reference_Values.Get_Min_HR() and
         Current_HeartRate <= Reference_Values.Get_Max_HR() and
         Buzzer.oclInstate(Off)))
```

Figure 3-15. Graphical representation of properties.

3.1.6.3.1 Non-functional requirements

The representation of non-functional requirements was not considered in our case study. Nevertheless it is possible to reason about the UML-based representation of such type of requirements:

- Modelling constraints on resources should be relatively easy. In fact, resources must be in s_v in order to include statements on resources in the requirements. This means that resources will be modelled as any other element in s_v.
- Modelling other constraints (e.g., on reliability) does not benefit directly from UML-based modelling. However the clear representation of functional requirements and of their elements provides a clear framework to talk about.

3.1.6.4 Benefits of the proposed approach

Although relatively precise and expressive, requirements described by means of UML and OCL are nevertheless not formal. From this point of view formal notations for requirements representation are still preferable, as they support formal reasoning and often allow formal verification of requirements specification. However the adoption of formal methods is considered difficult and expensive by industry, as long as it requires investments in learning and application, and does not provide immediate benefits. On the contrary, UML modelling can be adopted by all those (numerous) people who know UML but ignore more formal methods and notations. The proposed approach does not want to favour UML *instead of* formal methods, but rather to enhance the rigour of current methods, even in the context of UML usage. Moreover, the forthcoming formalisation of UML will allow the integration of formal techniques in UML-based development processes.

UML requirements are easy to convert into specifications. On their turn, thanks to UML's operational style, specifications provide a base for design and implementation (in general several classes belonging to the requirements model appear also in the code). I am not advocating seamless development [Mey88], nevertheless, it is quite clear that having written requirements in UML makes it easier to start a UML-centred development process. For instance, the availability of requirements written in UML makes the knowledge of the problem domain available to the designers, who can thus understand better the origin of the specifications they received.

Finally, UML-based models support traceability: concepts appearing in the requirements model can be tracked through the specification and design models to code elements. Given the great importance of traceability, we can conclude that UML–when correctly employed–provides a very positive contribution to software development practices.

3.1.6.5 Implications on development methodologies

The proposed approach can be applied in the context of the well-known UML-based development methodologies (e.g., RUP).

3.2 Requirements-driven, Model-based Development

The requirements-driven, model-based development approach presented here supports the development and testing of embedded systems. The starting point of the method is made up of requirements specifications. The method is illustrated using the modelling tool AutoFOCUS. The work supports evolving systems using tracing concepts and regression tests. Since a schematic derivation of test cases from a set of precise requirements is a major goal of this approach, the modelling language used to describe requirements is somewhat rigorous in comparison to other approaches.

The purpose of requirements engineering is at least threefold: The first goal is to describe a system, such that developers know what to implement; the second goal is to be able to verify that a system, once it has been built, delivers what it is supposed to; the third goal is to validate the engineered requirements against the user requirements (which are usually more abstract than the engineered requirements). Requirements have to be consistent, such that developers can build a system that satisfies all requirements, and requirements have to be precise in order to enable their verification once the system has been built. Requirements drive the development process by setting the goals and by verifying the results; hence it is of crucial importance that the requirements are not misleading but well-structured and precise. Therefore, within the requirements framework, the requirements have to be structured and prepared for the design and for verification.

In this part of the framework we describe a requirements modelling process (Section 3.2.2) that

- is based upon a requirements model,
- supports the integration into further development activities, and
- is precise enough for deriving test cases for verification.

Furthermore, the process is enabled for evolution; hence it can be used for reusing requirements and the requirements model. The process is based on a classification of requirements that is generic with respect to the used modelling language (see Section 3.2.3).

In Section 3.2.4 we develop a vision how the integration of requirements into a model-based development process could look like, and what features would be possible if precise and formal models, such as, e.g., those of AutoFOCUS, were used.

3.2.1 Relation to EMPRESS-Process

The empress process is influenced by the RUP. It provides several phases and disciplines to group different activities (see Figure 3-16).

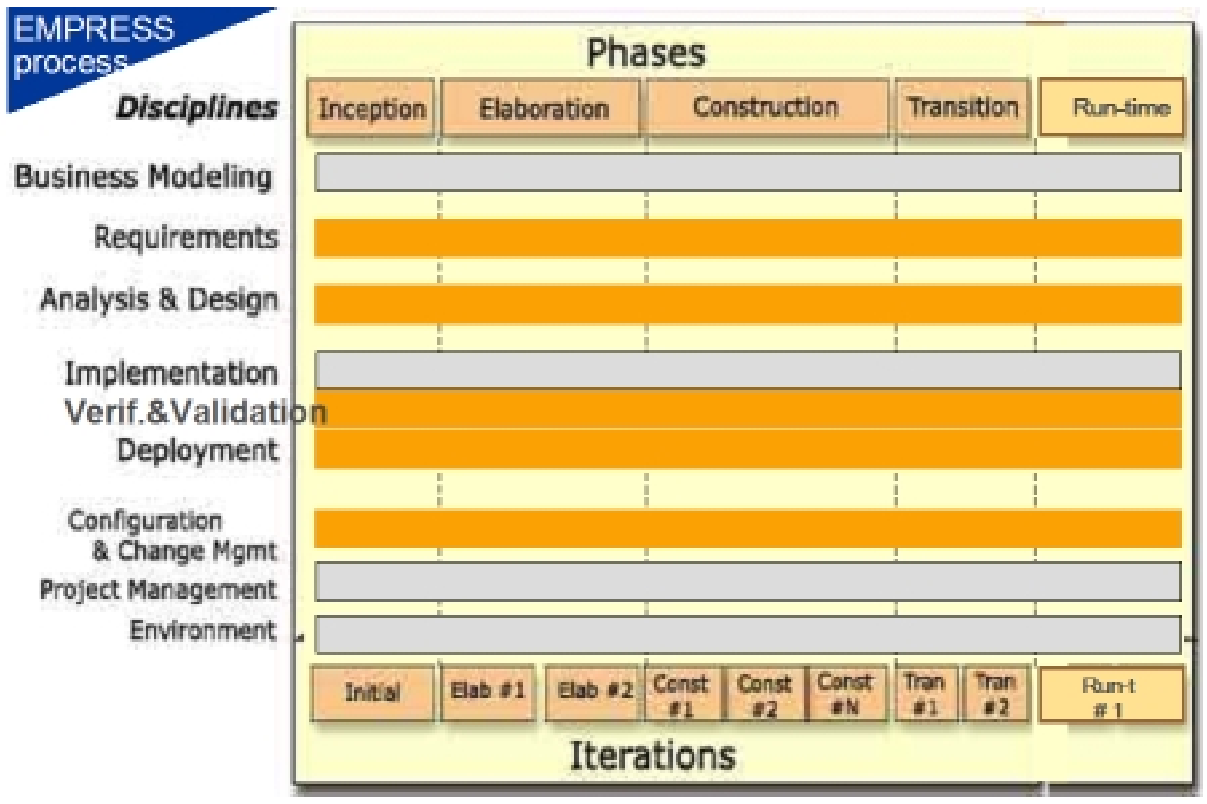


Figure 3-16: EMPRESS-Process

The requirements models are used within several disciplines and phases. The construction of the requirements models is an activity in the requirements workflow. This has to be done in early phases like Elaboration. After creating the requirements, they can be validated and used as starting point for model based-development. The requirements models can be used later for deriving test cases to ensure that the system has the appropriate level of quality (transition phase). Testing cannot only be performed in the transition phase but also in the construction phase, which has several benefits, especially in early error detection. Hence, the Validas contribution of requirements models can be integrated into the EMPRESS process as shown in Figure 3-17.

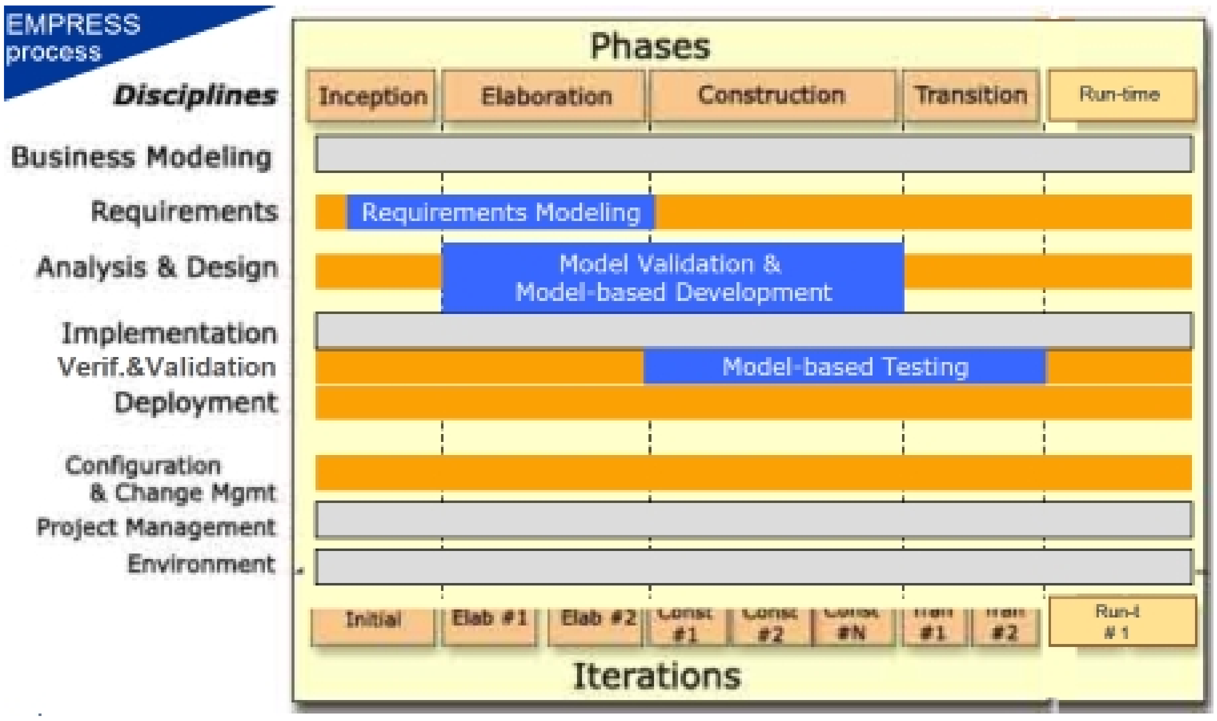


Figure 3-17: Validas Activities in EMPRESS-Process

3.2.2 Process: Modelling Requirements

The process of modelling requirements is not a meta-activity, but a concrete task during the development of a system. The created models should be re-usable for other variants of the system and for other product families. Using a suitable modelling approach, especially a suitable modelling language, can facilitate this. We do not suggest the usage of one particular language, but we prepare criteria for selecting the language. The model-based requirements structure in Section 3.2.3 is generic with regard to the modelling language. In the example in Section 3.2.4 we use the UML-RT-like modelling language of the research tool AutoFOCUS [AutoFOCUS].

In the description of the requirements process we have to deal with two situations:

- Situation 1: elicitation of system requirements (see Section 3.2.2.1)
- Situation 2: evolving requirements: from existing to new ones (see Section 3.2.2.2)

Both situations rely on a requirements model, hence we first define the purpose of a requirements model: A requirements model is a model that

- is abstract, i.e., that does not deal with implementation details,
- describes the interface of the system and, if desired, a coarse structure of the system,

- contains scenarios (interaction sequences) for different use cases, and
- is precise enough to test the system.

Ideally, requirements models should be sufficiently complete to be simulated and/or to be used for the generation of abstract test cases that have to be concretized for the verification of the implemented system. In Section 3.2.4 we present some ideas for the generation of test sequences and their usage.

3.2.2.1 Building Requirements Models

Generally speaking, the process of requirements engineering starts from initial user requirements and advances towards system requirements that are refined until they can be implemented. The modelling process can be performed in a bottom-up or top-down way. Bottom-up models are built from existing (atomic) models by putting them together, whereas top-down models start from an under-specified system and divide it into smaller models until an atomic component level is reached. In practice, a combination of both approaches will usually be the case. For the first requirements model, we suggest a top-down modelling approach, especially, because the requirements models usually are very abstract in the beginning and require further concretization.

Another issue is the relevant aspects to be included in the model. Since more and more functionality in modern systems is based on software, we focus of software requirements of the system.

The overall process consists of the following steps

- requirements structuring, especially identification of software requirements (design parts) and tests (sequences or cases)
- model-based classification of requirements (see Section 3.2.3.1), especially separation between design and test requirements (test cases/scenarios)
- development of the functional model for the functional requirements (see the rest of this section)
- validation of the requirements model (i.e., can the test requirements be fulfilled by the requirements model of the design?)
- generation of further test cases for the system (from the requirements model)
- development of a system from the requirements model
- verification of the developed system against the generated test cases

From the model-based classification of the software design requirements a first model can be generated (see Section 3.2.3.1) that has to be refined in order to build a precise and to some extent complete model. The generated model frame can serve as a starting point for development of the requirements model. The process of building requirements models is depicted in Figure 3-18. It shows the refinement process from the generated model frame (Note: this process can also be used to build a model from scratch) towards a precise requirements model that can be used as starting point for the development and as a basis for the generation of test sequences for requirements verification.

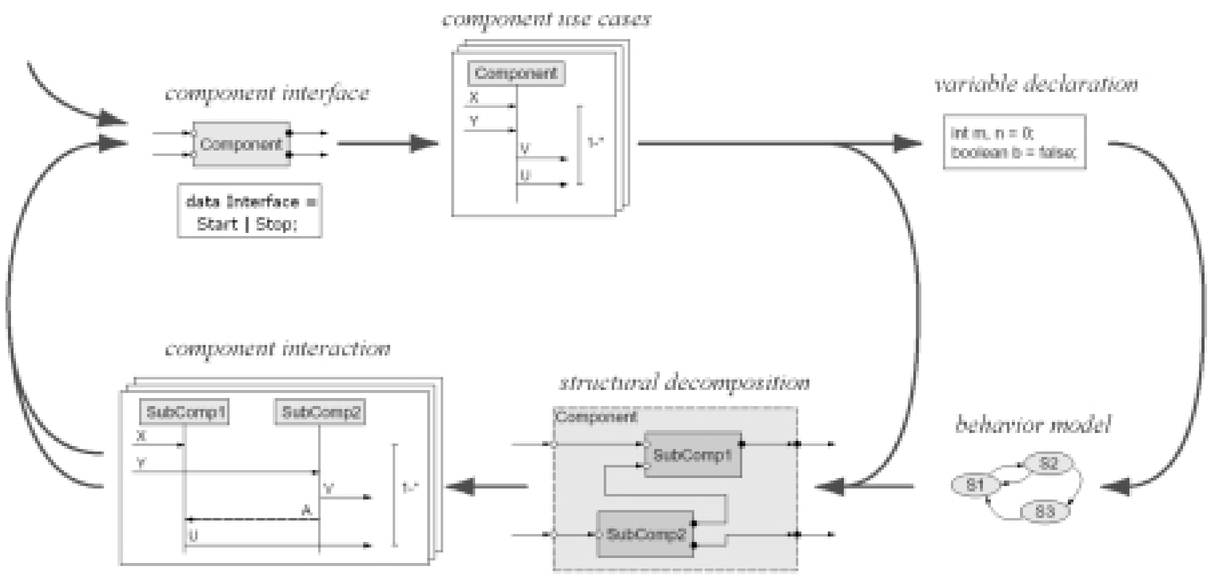


Figure 3-18: Process of Modelling Functional Requirements

The process starts with modelling the system interface (communication points and their data types). Note that the elements of the data types can still be abstract, like “EnterData” or “StartEngine”. Based on this interface first use cases (sequences) can be specified. The next step consists in modelling the behaviour (using variables and state transition diagrams). Alternatively, the modelled component can be refined by a structural decomposition into subcomponents. The interaction of the subcomponents is described again using sequences that distribute the global black-box interaction among the subcomponents. Now the next iteration of the process starts for the subcomponents with a description of their interfaces.

After the requirements model is complete, the test cases for the verification of the requirements have to be derived. This can be done manually (e.g., by creating sequences or by simulating the model), or partially or completely automatic. The degree of automation depends on the testing requirements (coverage, etc.) and the complexity of the models.

3.2.2.2 Evolving Requirements

The process for evolving systems shall answer the question of reusing requirements and adopting existing requirements. During this evolution process the extra effort spent for requirements modelling during the first phase pays off. The requirements models can be reused (assuming that they are structured in a modular and reusable way, especially with a modelling language that supports reuse), and the test cases can be taken over as well. For the system development it is important that an association between the requirements models and the system implementation is maintained, such that also the corresponding implementations can be reused.

The process of evolving requirements essentially deals with the development of requirements (from existing requirements) to new requirements. The main steps are reusing and adopting requirements. With the proposed approach of Section 3.2.2.1 we assume the existence of the requirement models that can be “evolved”.

If the requirements change their corresponding requirements models have to be adapted. There can be different kinds of model changes depending on the type of requirement change:

- adding a new scenario (test case model) to the model
- changing an existing test case of the test model
- adding new model elements

- changing model elements

Of course, evolution of requirements imposes re-validation of the requirements models and testing of the conformance between the implemented, “evolved” system and the requirements model (re-verification).

In order to improve the evolution process, the steps for propagating the changes should be automated as far as possible. While linking requirements and models on the requirements side is a well-automated process, the generation of test sequences is still more or less manual work and therefore a crucial point for the remaining time of the EMPRESS project. In Section 3.2.4 we develop visions of an automated process for that purpose.

3.2.3 Model-based Requirements Structures

In this part of the framework we propose a requirements structure that is generic with regard to the actual modelling language and can be used for many modelling languages (Section 3.2.3.1). In addition, we develop criteria for the selection of appropriate modelling languages that are especially suited for the evolution of requirements.

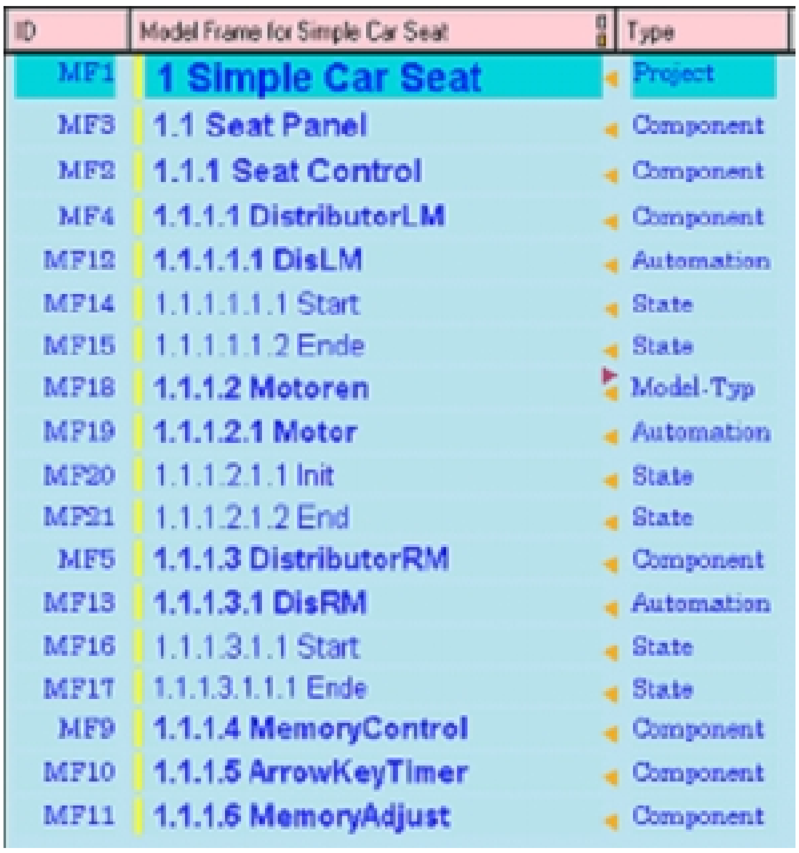
3.2.3.1 Classifying Requirements for Models

Within the overall requirements framework (see the overview Section in this document), there are several classifications for requirements for different purposes. The classification presented here is for building a model frame that can serve as basis for a requirements or an implementation model.

The classification principle is simple: Every functional requirement is assigned with a meta-model information that describes the kind of the model element to which the requirement shall be assigned. The assignment is made as part of the requirements engineering process to classify the requirements into tests and design elements. Further classification using the meta-model is the first step of the design and will be carried out by the designer. The meta-model information depends on the chosen modelling language. Typical meta-model information kinds are: state, message, transition, class, function, etc. For the assignment of this information an additional requirements attribute is necessary (in the general model for requirements). An example (within the DOORS tool) for this information is depicted in Figure 3-19.

The model elements can be sorted according their classification. However, more important for large models is the structure of the requirements that shall be as close as possible to the structure of the models (assuming that the used modelling language supports hierarchic models).

Based on the structured models, the next step is the generation of model frames (including the important links to the requirements) for further refinements within the modelling tool.



ID	Model Frame for Simple Car Seat	Type
MF1	1 Simple Car Seat	Project
MF3	1.1 Seat Panel	Component
MF2	1.1.1 Seat Control	Component
MF4	1.1.1.1 DistributorLM	Component
MF12	1.1.1.1.1 DisLM	Automation
MF14	1.1.1.1.1.1 Start	State
MF15	1.1.1.1.1.2 Ende	State
MF18	1.1.1.2 Motoren	Model-Typ
MF19	1.1.1.2.1 Motor	Automation
MF20	1.1.1.2.1.1 Init	State
MF21	1.1.1.2.1.2 End	State
MF5	1.1.1.3 DistributorRM	Component
MF13	1.1.1.3.1 DisRM	Automation
MF16	1.1.1.3.1.1 Start	State
MF17	1.1.1.3.1.1.1 Ende	State
MF9	1.1.1.4 MemoryControl	Component
MF10	1.1.1.5 ArrowKeyTimer	Component
MF11	1.1.1.6 MemoryAdjust	Component

Figure 3-19: Requirements Classification

More details on the classification process can be found in [TUMaster] and [BS00].

3.2.3.2 Criteria for Modelling Languages

Within the description of the process, we had several statements, such as “modular models can be reused”. In this section, we collect the requirements for the modelling language imposed by the process. In Section 3.2.4 we will then develop the vision of the process that is based on the UML/RT-like modelling language of the tool AutoFOCUS, which fulfils these requirements to a large extent. The important selection criteria are:

- **Hierarchy**: The modelling language shall support hierarchical descriptions. Examples are states in states (“substates”) or a component refined by subcomponents. This is important for breaking up models into parts (top-down-approach).
- **Modularity**: Ensures that the parts of the model are independent from each other and can be reused for building a system (without side effects, bottom-up-approach). Global variables, for example, violate the modularity principle.
- **Clarity**: The modelling language shall be clear and understandable.
- **Precision**: The models shall be precise, such that the semantics/behaviour is clear.
- **Abstraction**: The modelling language shall support abstract models, as well as the concretization into more implementation-oriented models.
- **Reuse of models**: The modelling language shall have explicit constructs for reusing models. This will keep the designed models simple and will help to avoid unnecessary redundancy.
- **Benefits for further processes**: The models shall have a benefit for the further

development process (not only for documentation purposes). There are many possible benefits of models for the development process:

- **Validation**: The models shall be accessible for analyses to validate them. Important validation steps are:
  - **Non-functional requirements**: can be validated using the models, for example modelling guidelines can be checked, or usability can be simulated.
  - **Simulation**: can show the reaction of the system on some inputs (sequences). Simulation allows debugging of models on the model level.
  - **Analysis**: Models can be analyzed, for example to compute the load for a communication channel, or to estimate the number of test cases needed to cover the generated code. Some models support coarse estimations of worst case time behaviour or the calculation of required implementation size.
- **Adequacy**: The models shall be adequate for describing or solving the problems: It shall be easy to model (without using many model elements) requirements that can be easily formulated using natural language or programs.
- **Consistency**: The modelling language shall support the formulation and checking of consistency conditions, for example, to check project-specific modelling guidelines.
- **Documentation**: The modelling language shall be well-suited for documenting the system. This requires the ability to add comments to the model elements or to include textual descriptions via references.
- **Code generation**: A core feature, especially if detailed design and implementation models have to be built. The code generation should ideally result in production code. Furthermore, tracing from the code to the model should be possible.
- **Test automation**: Support for the generation of test drivers or test sequences from models should be available, such that the models can also be used for testing the system.

- **Standards**: The modelling language should adhere to standards as far as possible.
- **Tools**: The modelling language shall have tool support. The availability of tools is often crucial for the chosen variant of the modelling language.

Formal development approaches fit well to many of these criteria, however, they are mostly not well-accepted in practice due to a variety of reasons, ranging from insufficient (practical) scalability for large development projects to often cryptic, non graphical description techniques[1].

3.2.4 Vision: Evolutionary Development Process using AutoFOCUS Requirements Models

In this part of the requirements framework, we illustrate the potential power of the requirements modelling approach using the graphical and formal modelling language of AutoFOCUS. We do not present a complete example, just some images and process steps that are possible (and to some extent even now available within AutoFOCUS).

[1] See http://www.afm.sbu.ac.uk/ for an overview over formal methods.

In this scenario, we use the AutoFOCUS models for requirements modelling, but not for the generation of target code. This allows a more abstract modelling style, which is more understandable and less work compared to an implementation model.

This process example for requirements evolution starts with a requirements model that is already assumed to be validated and implemented within an existing system. The model uses stereotypes (stereotypes are comments with a semantic meaning, in our case: /* Req <id> */) to represent the links to the requirements within the model. These stereotypes are important for later use of the requirements. The system structure is modelled using AutoFOCUS system structure diagrams. An example is depicted in Figure 3-20. Within the system structure diagrams the requirements that have been classified as Component, Variable, Port or Channel are visible at the corresponding elements. In the example of Figure 3-20 the variable "KundeNr" has been introduced for the requirement 3.1.2 "store Kunde".

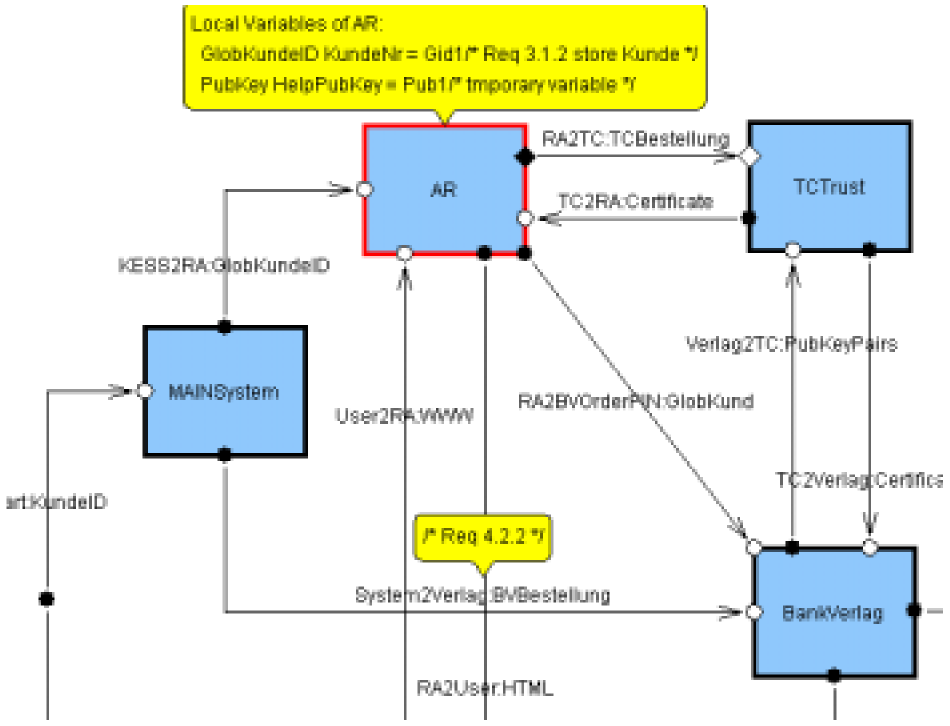


Figure 3-20: System Structure with Requirements Links

The behaviour is modelled using state transition diagrams, as for example in Figure 3-21.

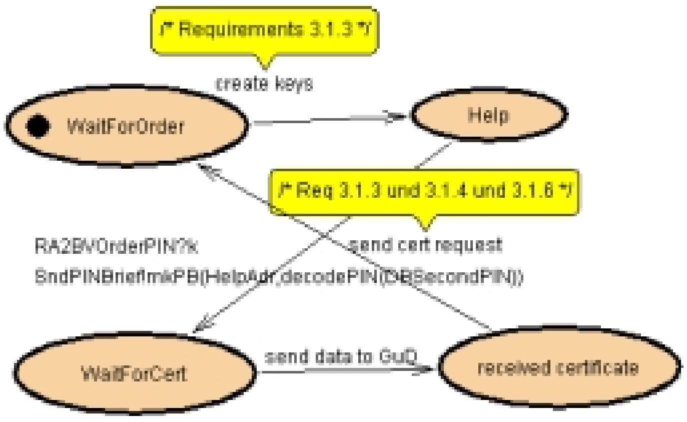


Figure 3-21: Behaviour of a Component with Requirement Links

The use cases and test sequences are modelled using sequence diagrams. They can be exported into text files, which can be used for test driver generation and regression testing (see an example in Figure 3-22).

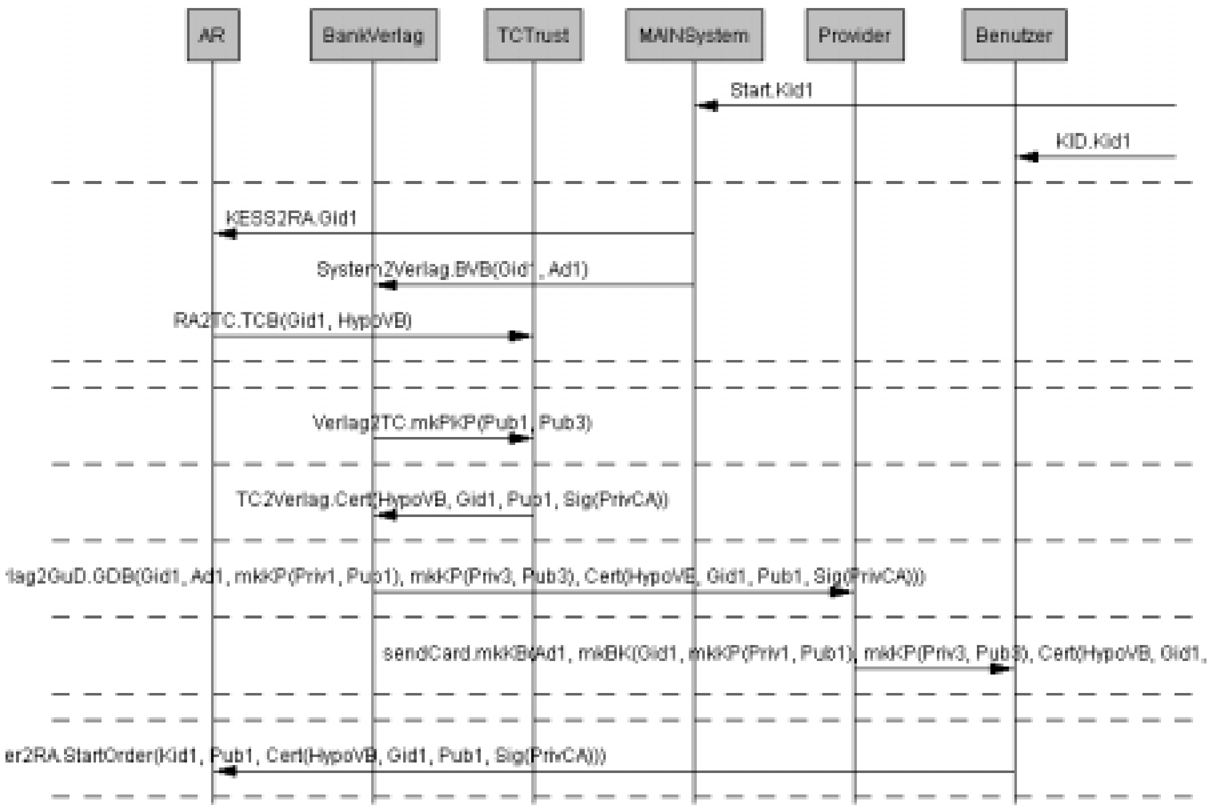

AR
BankVerlag
TCTrust
MAINSystem
Provider
Benutzer
Start.Kid1
KID.Kid1
KESS2RA.Gid1
System2Verlag.BVB(Gid1, Ad1)
RA2TC.TCB(Gid1, HypoVB)
Verlag2TC.mkPKP(Pub1, Pub3)
TC2Verlag.Cert(HypoVB, Gid1, Pub1, Sig(PrivCA))
lag2GuD.GDB(Gid1, Ad1, mkKP(Priv1, Pub1), mkKP(Priv3, Pub3), Cert(HypoVB, Gid1, Pub1, Sig(PrivCA)))
sendCard.mkKB(Ad1, mkBK(Gid1, mkKP(Priv1, Pub1), mkKP(Priv3, Pub3), Cert(HypoVB, Gid1,
er2RA.StartOrder(Kid1, Pub1, Cert(HypoVB, Gid1, Pub1, Sig(PrivCA)))

Figure 3-22: System Interaction Sequence

Based on such models *evolutionary steps* can be performed using the formal basis of the models:

If a *new scenario* is required, it has to be validated that the scenario (use case / sequence) is possible. This can be done using simulation (in simple cases) or in matching the sequence against the model. The new scenario will be stored in a text file and added to the regression test suite for the model. The regression test suite is a set of test sequences that cover the requirements model and can be checked by executing the model for validity.

If a *new requirement* for the model is added, the model has to be adapted accordingly and the main question is the consistency of the requirements model with the new requirement. After the addition of the new requirement into the requirements model, the regression test suite (that tests all test models automatically) is applied to check the consistency with the old model. The differences (errors) resulting from the new, adapted model are reported (if the new requirement is inconsistent). Errors can be caused by faulty changes of the model and require that the model is corrected. Another source of errors is that an old test case conflicts with the new requirement. This requires solving the conflict for example by changing/adopting the old requirement.

Using code *coverage measurement* the quality (in the sense of completeness) of the current regression test model suite is evaluated. If uncovered code (generated from the model) exists, this means that not the complete model (that models the requirements) has been covered from the current set of tests, hence not all requirements are covered from the tests. Therefore, further test case models have to be created (automatically or manually) that cover the remaining requirements.

For *traceability* to the implementation the effects of the changes in the model shall be printed out in a list and will be given to the system designer in order to change their system design.

After the changes are incorporated into the requirements model, the model can be *analyzed* to verify critical aspects like system size, communication load or schedulability of the model. If the model does not fit to the current hardware, hardware parameters (or the system deployment) have to be changed and the evaluation of the architecture can restart again until the deployment of the system can be applied.

Some of these process elements are ready for application (the requirements models, the regression test suite, the code generation, the coverage measurement, and parts of the automatic test sequence generations), some of them have to be tailored by need, for example to use specific scheduling strategies or hardware models.

3.2.5 Experiences from Demonstrators

Validas AG has shown the benefits of the approach on the demonstrator applications of SIEMENS and DaimlerChrysler. The results are described in Deliverable 5.2

4 Integration of classifications, structuring and process models

4.1 Introduction

In the development of software with evolving requirements, activities of requirements engineering and management are present through the whole software development process and affect most of the actors involved; for example, a change in the requirements may affect the current implementation status as well as the test cases.

To support the collaboration, an integration of the requirements discipline with other disciplines has to be achieved (see [Kru00] for details on disciplines). On the one hand, in the requirements discipline requirements are captured and structured and thereby the other disciplines are provided structured views on the specification. On the other hand, the other disciplines supply tracking information by establishing a relationship between their work products and the requirements. The presented approach is inspired by a requirements engineering method for complex COTS software (cf. [Dei01]) and offers a central data structure for requirements as well as methods to support the tasks listed above.

First, a generic data structure will be introduced (section 4.2.1) together with a couple of methods to refine this generic structure to one's specific needs (section4.2.2). Secondly, the proposed generic structure and the refinement methods will be used to develop a data structure dedicated to the domain of embedded real-time systems (section 4.3). The next section (4.4) will point out, how to continue this refinement process in order to adapt the data structure to more specialized application domains (e.g. automotive embedded real-time systems). Finally the integration of this data structure into a software-engineering process, especially its use by different project roles in the phases of the development process (e.g. inception, elaboration, construction, transition) is covered in section 4.5.

4.2 The Generic Data Structure and Refinement Techniques

In this section, an abstract data structure will be introduced, together with a couple of methods to adapt this generic structure to specific needs.

There are three reasons for developing a specific structure out of a generic structure, instead of presenting the specific structure at once:

- it demonstrates the usage of adaptation techniques, which makes it easier for users to continue this process
- it reveals the decisions that led to the specific structure
- it makes it easier to explain the complete structure by introducing concepts step by step, increasing the degree of specialization

4.2.1 The Generic Data Structure

For the description of the data structure in its various stages of refinement, class diagrams of the Unified Modelling Language (UML) will be used. The elements of this notation are:

- Classes (including Attributes)
- Associations
- Aggregations
- Inheritance

For more information about UML, see [Fow99].

The generic data structure is the highest abstraction level. The basic element of the data structure is the “development product” (see Figure 23).

<<abstract>>
Development Product

Figure 23: Generic Structure

The abstract class “development product” has no special semantic meaning and only has the purpose to serve as a starting point for the formal adaptation process in this chapter. The adaptation methods applied in this chapter are two techniques of refinement which will be introduced in the following section.

4.2.2 Refinement Techniques

According to Cambridge Dictionaries [Cam03] refinement means to improve something by making small changes.

Though this definition opens a big variety of possible refinement techniques, this chapter focuses on two refinement methods that are well defined and easily manageable, but still sufficient for adapting the data structure to specific needs:

- Specialization
- Tailoring

Specialization means that classes are extended (likewise the mechanism of inheritance) to map a more specific view on the application area to the data structure. For example, a general element “DesignArtefact” in Figure 24 might be rendered more precisely as “Interfaces” and “Components”:

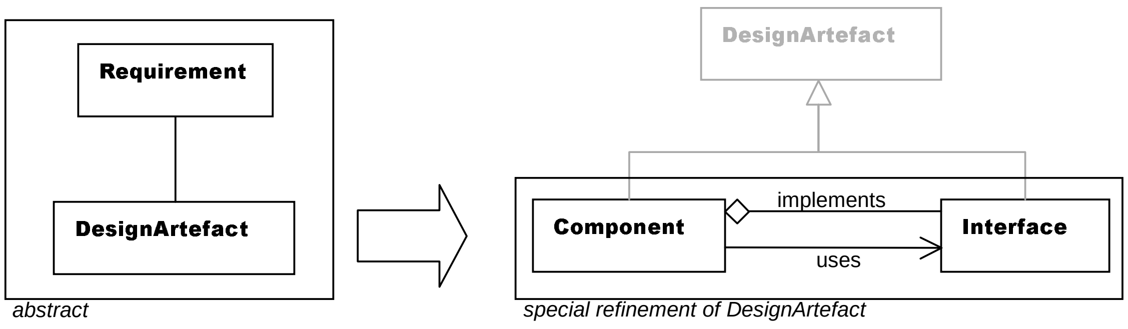


Figure 24: Specialization of DesignArtefact

To increase the readability of the specialized model, it is not necessary to redraw all associations inherited from higher abstraction levels. As shown in Figure 24 the association to the class “Requirement” is implicitly given for “Component” and “Interface”.

The term “tailoring” means to cut elements from the data structure. In some special application domains it is possible to abandon particular elements of the data structure, since the activities working on these elements can never occur. The tailoring of UML models means removing classes and associations respectively:

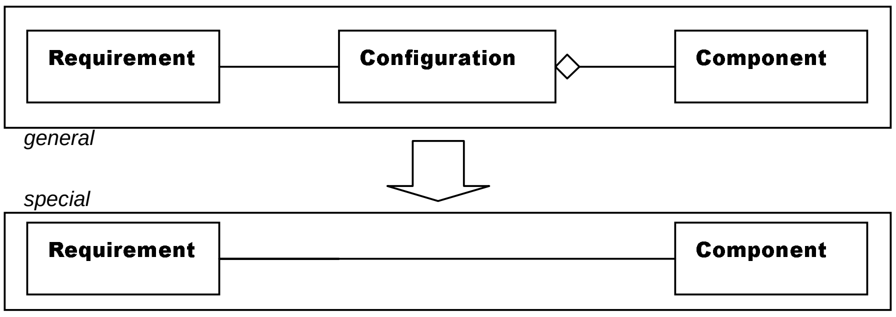


Figure 25: Tailoring of the model

If a class is removed, it may be necessary to redirect associations that have been connected to that class. An example is given in Figure 25, where after the removal of class “Configuration” the association between “Requirement” and “Component” has to be retained.

4.2.3 Refinement Steps applied to the Generic Model

In the remainder of this chapter, the generic model (presented in section 1.2.1) will be refined using the techniques from section 1.2.2, in order to adapt the model to the domain of embedded real-time systems. Therefore, the following refinement steps have to be performed (see Figure 26):

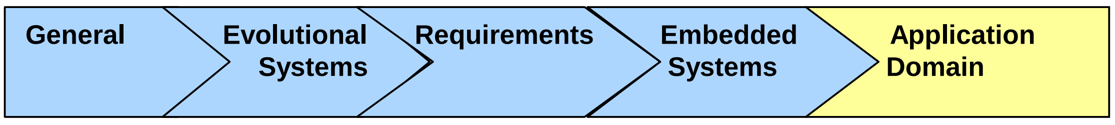


Figure 26: Refinement Steps of the Data Structure

In the first step the model will be refined in order to be able to trace changes and document their reasons by introducing the concepts of history and context. The second step is an adaptation to the needs of requirements engineering and requirements management by offering elements to capture, structure and analyze requirements and by providing connections to other disciplines. The next level of detail deals with properties of the real-time embedded systems domain. The result of section 4.3 is a model for requirements engineering and management in the domain of evolving real-time embedded systems.

Since the applications in the field of embedded systems are very heterogeneous, the model given in section 4.3 has to be quite general. Therefore a further step of refinement will be necessary, before the application of the model in practice may happen. Section 4.4 contains different refinement suggestions and examples for specific domains.

4.3 The Conceptual Model

The following subsections describe the model on different abstraction layers as shown on Figure 26. The description is supported by an example of a hypothetical window lift system. It resembles a sample specification of a “Türsteuergerät” (Door Control System) of DaimlerChrysler [Hou01]. The example is simplified and slightly changed:

Window Lift Controller

The task is to develop the software for a window lift controller that allows the comfortably lifting and lowering of the side windows of a car. Every door has a controller which is connected to push-buttons, an electric motor and a CAN-bus of a car. Each controller can be turned on or off together with its peripheral devices. The movement of the windows is triggered by push-buttons. As long as a push-button is pushed, the corresponding window is moving up or down.

In the driver-door are push-buttons for all windows (4 in sedan or 2 in open-topped). The remaining doors (non-driver) have one push-button for their window. If the child-lock is activated, the push-buttons in the back doors of the car are deactivated.

4.3.1 The Conceptual Model for Evolution

The top abstraction level of the generic model (see section 4.2.1) is refined to a conceptual model in order to capture and trace changes on development products (see Figure 27):

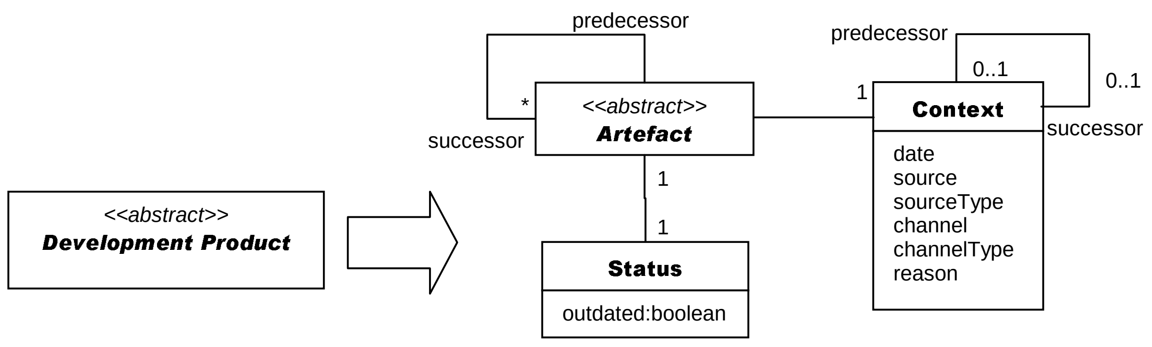


Figure 27: Conceptual Model for Evolution

At this level of detail, five elements are introduced:

- a class “Artefact”, which represents all products that occur in a development process
- an association “predecessor/successor” between artefacts, which allows to record change histories
- a class “Status” indicating whether an artefact is valid or outdated
- a class “Context”, which allows capturing the origin of an artefact
- an association “predecessor/successor” between contexts, which forms a qualitative timeline for the whole set of artefacts

The conceptual model for evolution may be used to describe any product (artefact) that occurs in a development process (e.g. single requirement, group of requirements, release). At this degree of abstraction no distinction between the different product types is made; the necessity of a further refinement is depicted in the model by defining "artefact" as an abstract class.

In an evolutional process artefacts may change several times. For purposes of traceability the changed artefact does not overwrite the old one. The new version rather replaces the old version by setting a link from the old one to itself using the successor association and the status of the old version is set to “outdated”. Note that the removal of an artefact is treated as a special case of a change operation. A removed artefact may not have a successor, but its status is outdated. By this means it is possible to keep track of the changes of an artefact.

Each artefact (each version of an artefact, respectively) is furnished with a link to a context, which mainly describes the circumstances of its creation, change or deletion. Several artefacts can be changed within one context; thus a context groups artefacts. The class “context” has following attributes:

- *Date*
  The date of the change is an important criterion for the analysts to check how up-to-date a change is. Further on there is the possibility to implicitly gather context information associated with that date (e.g. the business situation of the company)
- *Reason*
  Change actions are usually caused by a reason. This information can be held in the context. The use of this field may force the developers to explicitly write down their decisions. This attribute may range from a small item up to a complex document.
- *Source*
  If a requirement is created or a change is performed the persons who initiated this act should be registered. The possibility of associating the actors to an action allows finding persons who are responsible for features or changes of the system.
- *Source Type* (Role)
  People representing multiple roles are working on the model. While the physical responsibility is contained in the attribute “Source” (e.g. “Mr. Meier”), the logical responsibility (e.g. “Customer”) may be stored here. This attribute may be used to check access rights to the model. Typical source types for example are “Requirements analyst” or “Designer”.
- *Channel*
  A channel describes the medium which brought the change into the model. It is one factor among others that have an influence on the properties of a requirement. For example, a requirement gained in a chat may be less credible than one contained in a contract.
- *Channel Type*
  Channels can be categorized in regard to their quality and their binding character. This classification is done with this attribute.

Every context (except the first one) has a predecessor that references the change action before. Thereby, the contexts form a qualitative timeline. With this structure, change actions in the development process can be traced.

An example for a context of a requirement can be found in Figure 33 in section 4.3.2.1.2. The change occurring in this figure is caused by complaints of customers (reason) and handled by the usability-lab (source). The results are gained in discussions during a workshop (channel).

4.3.2 Conceptual Model for Requirements

This section describes the conceptual model for requirements, which is refined from the conceptual model for evolution introduced in the previous section, using the refinement techniques from section 4.2.2. The presented model is a refinement of the abstract class

“Artefact” to several specific artefacts. These special artefacts are:

- The artefact “Requirement” is the representation of the term requirement.
- The artefact “Aspect” groups requirements according to their structural or causal dependencies.
- The artefact “Configuration” maps requirements to a specific product, which has to realize them.
- The artefact “DesignArtefact” represents an abstract entity for design activities.
- The artefact “Release” is an entity for planning the realization of requirements for a specific design artefact.
- The artefact “Configuration Release” is used to plan the realization of individual architectures.

Since each of these artefacts is derived from the abstract class “Artefact”, it inherits its “predecessor”-relation, status as well as the association to a context.

There are two further artefacts defined by the presented conceptual model for requirements. These are “Variation” and “VariationType”, which are derived from the artefacts “Requirement” and “Aspect” respectively. Together they define contradictory requirements, which describe alternative architectures and design decisions, comprised by the entire set of requirements.

Figure 48 demonstrates the conceptual model for requirements. Differently coloured rectangles in the background group artefacts according to the project disciplines, which they are intended to support.

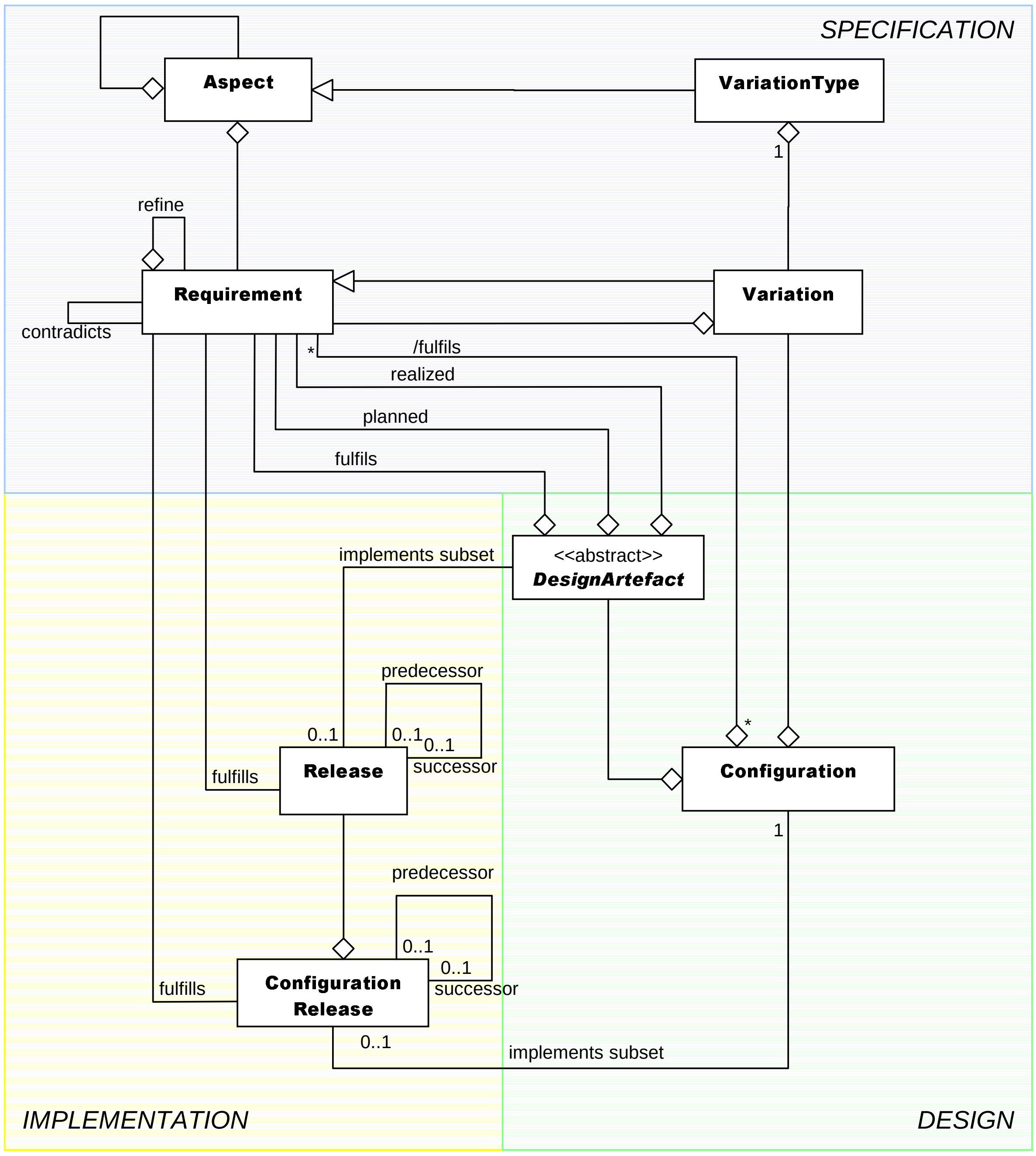


Figure 48: The conceptual Model for Requirements.

The four artefacts “Requirement”, “Aspect”, “Variation” and “VariationType” serve for the purpose of capturing, structuring and analyzing requirements in the requirements engineering discipline. They are described in section 4.3.2.1. The rest of the artefacts serves for tracing purposes in the requirements management discipline. Section 4.3.2.2 introduces the interface of the presented model to the design discipline. The interface is built by the “DesignArtefact” and “Configuration” and allows deploying a mapping between requirements and design artefacts. This mapping supports the traceability of requirements to design. Finally section 4.3.2.3 relates the model to the implementation discipline. One of the implementation discipline’s tasks is the planning of realization order of requirements for every design product. The realization order can be adhered in an instance of the presented model using the artefacts “Release” and “Configuration Release”. This allows the tracing of requirements in the implementation discipline.

The description of every artefact in the following sections will motivate its existence in the model, identify the real-world phenomena, which should be mapped to it and define its relationships to other artefacts. In order to demonstrate the advantages of applying the presented model, different constraints and criteria are formulated for and between artefacts. In the remainder of this chapter, formally noted constraints are captured in boxes. They can be checked automatically in order to guarantee a low-level consistency of the specification document and to release as far as possible the users of this model from routine, time-intensive and error-prone activities.

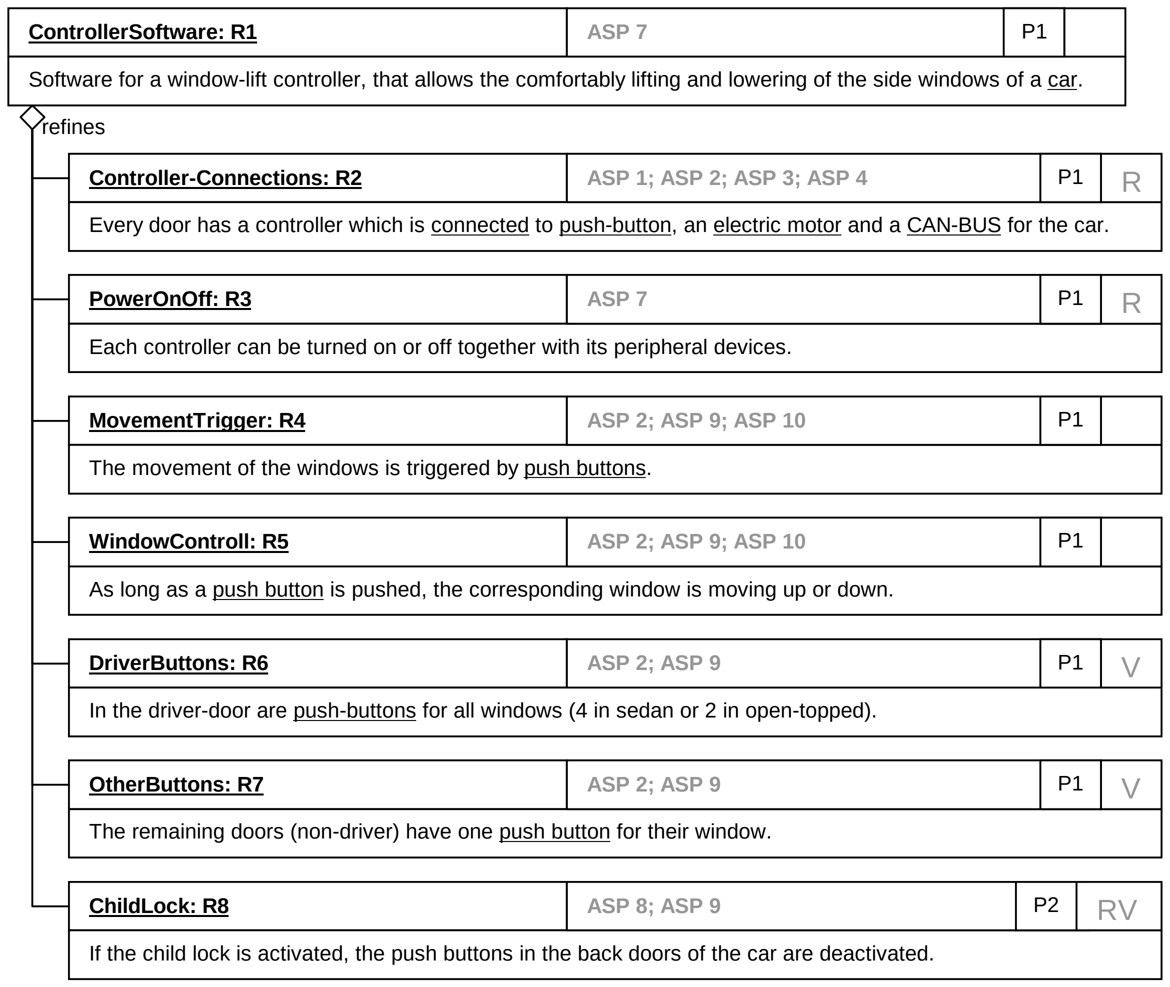
4.3.2.1 Specification View

The most important goal of the developed model is the support of activities, which aim the building of a consistent and user-friendly specification. Those activities can be mapped to the documentation and analysis phases of the requirements engineering. Another important goal is the support of the change management. The model introduces concepts, which try to reduce the search field for possible impacts of a change event.

These central aspects of the conceptual model will be considered in the following subsections, structured according to the different artefacts.

4.3.2.1.1 Requirement

Requirements describe the measurable properties of a product to be developed. They have to be gathered, systematized and validated in respect of consistency and completeness. During all of these processes different relationships between requirements have to be comprehended. These are among others contradiction- and refinement-relations. Another task is to consider different attributes, which describe the priority, changeability and other characteristics for a particular requirement. Requirements are represented by the artefact “Requirement” in the model.



ControllerSoftware: R1	ASP 7	P1	
Software for a window-lift controller, that allows the comfortably lifting and lowering of the side windows of a car.			

refines

Controller-Connections: R2	ASP 1; ASP 2; ASP 3; ASP 4	P1	R
Every door has a controller which is connected to push-button, an electric motor and a CAN-BUS for the car.			
PowerOnOff: R3	ASP 7	P1	R
Each controller can be turned on or off together with its peripheral devices.			
MovementTrigger: R4	ASP 2; ASP 9; ASP 10	P1	
The movement of the windows is triggered by push buttons.			
WindowControll: R5	ASP 2; ASP 9; ASP 10	P1	
As long as a push button is pushed, the corresponding window is moving up or down.			
DriverButtons: R6	ASP 2; ASP 9	P1	V
In the driver-door are push-buttons for all windows (4 in sedan or 2 in open-topped).			
OtherButtons: R7	ASP 2; ASP 9	P1	V
The remaining doors (non-driver) have one push button for their window.			
ChildLock: R8	ASP 8; ASP 9	P2	RV
If the child lock is activated, the push buttons in the back doors of the car are deactivated.			

Rx = Requirement x; ASP x = Aspect x; Px = Priority x; R = Refinement necessary; V = Variants necessary

Figure 49: Requirements for Window-Lift-System

Each "Requirement" artefact encapsulates exactly one requirement, given in any representation form, e.g. text, formula, image, use cases etc. Furthermore it captures the attributes and defines relationships between different requirements mentioned above. Figure 49 demonstrates some requirements of the window-lift-system example.

Requirements are often given by customers combined with additional information. It can be an informal description, intended to clarify the requirement, or some information for handling the requirement. For the current version of the conceptual model the following attributes are suggested:

- *Priority*
  The priority can constitute the implementation order or simply the importance of requirements inside of one specification document. Figure 49 shows implementation priorities for every requirement in the second right column. The priority scale in this example has two states: high (P1) and low (P2).
- *Change probability*
  The evolution of requirements affects the design and implementation of a product intensively. The (heuristic) estimation of change probability (e.g. [Low, High]) helps to prepare for and to consider the possible change impacts before the change happens. These attributes are kept together with the requirement, which they classify, in the artefact "Requirement" (see R9 on Figure 51).
- *Refinement necessary*

Some requirements in the data structure may be too general or abstract. If such a requirement is detected, it can be marked for a refinement. With the help of this flag, the requirements to be refined can be easily found again. Refinement of requirements is discussed in the next paragraphs.

- *Variation candidate*
  Contradicting requirements are allowed, if they exist in different products of a product line. These requirements can be marked as candidates for variations (see section 4.3.2.1.3 for details).

The status for requirements is captured by the class “ReqStatus” from Figure 50. It is derived from “Status”, presented in section 4.3.1. The requirements gathered during the elicitation are inserted into an instance of the conceptual model, marked as “proposed”. The decision whether they will be included into specification or not has to be met afterwards. Depending on this decision, a requirement becomes “approved” or “rejected”. Rejected requirements are not modelled explicitly here. The affected requirements are just deleted, by setting the artefact’s status to “deleted”. An approved requirement can be deleted also.

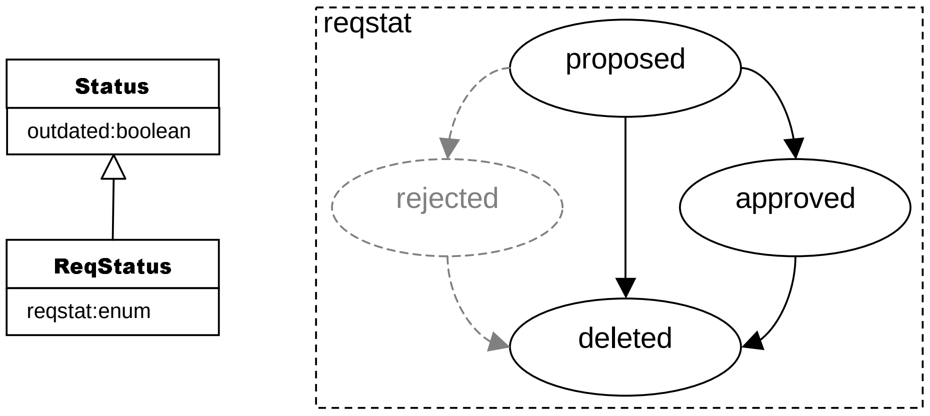


Figure 50: Refinement of Status for Requirement Artefact

There exists a homomorphism from the status of a requirement described above to the status of an arbitrary artefact, represented by the attribute “outdated”, inherited from development product “Status”. The “outdated”-attribute is set to “true” if and only if the “reqstat”-attribute has the value “deleted”.

In the rest of this chapter, only requirements with the status “approved” will be treated. Requirements with the status “proposed” and “deleted” can not participate in any of the relationships presented here. If an approved requirement was deleted, its associations to other artefacts have to be removed too.

During the negotiation phase of the requirements engineering process contradictory requirements can be formulated by customers. The existence of one contradiction in an instance of the model makes the whole instance inconsistent. These contradictions are expressed in the model by the association “contradicts”. The contradictions documented by this relation have to be resolved manually by the requirements engineer before further work on the model can proceed. This demand prevents the propagation of inconsistencies in the space of negotiated requirements. A requirement can participate in several “contradiction” dependencies.

Another relationship between requirements is called refinement. The refinement of requirements is acquired by the association “refine” in the model. During the elaboration phase the requirements are continuously stated more precisely (refined), until a specific refinement degree is achieved, which allows a clear and detailed specification of the product to be developed. For example, the requirement R1 in Figure 49 is refined to requirements R2-R8. The letter “R” to the right of a requirement in the figure signalizes the need of further refinement. Fragmentary or wrong refinements are a frequent source of inconsistencies in specification documents. This fact motivates the acquisition of a refinement procedure in form of an abstraction/refinement-hierarchy in the model. The hierarchy is established between any original requirement (abstraction) and all of its refinements. One requirement can be refined to a set of requirements. The intersection of refinement sets of several requirements needs not to be empty. This means, that a requirement can have more than one abstraction. For example the requirements R2, R4-R8 from Figure 49 could be refined to a requirement for existence of a push-button interface in the window-lift-system (R10), e.g. R10 would have R2, R4-R8 as its fathers in the refinement-hierarchy.

The refinement of the conceptual model, introduced in section 4.2.2 should not be mistaken for the refinement of requirements. In the following sections the term “refinement” refers to the requirements’ refinement defined here.

Another motivation for a refinement-hierarchy is its possible use for bordering the impact of change. A change of a requirement can influence its father, its descendants as well as its siblings in the hierarchy. Consider for example the requirement R9 refined from R3 as shown on Figure 51. Suppose R9 is replaced by a new requirement R9’. The change of R9 doesn’t influence its parent R3. So the search process of change impact can be stopped there. However the impact cannot be assumed as fully acquired. Other mechanisms for this purpose will be introduced in the following sections.

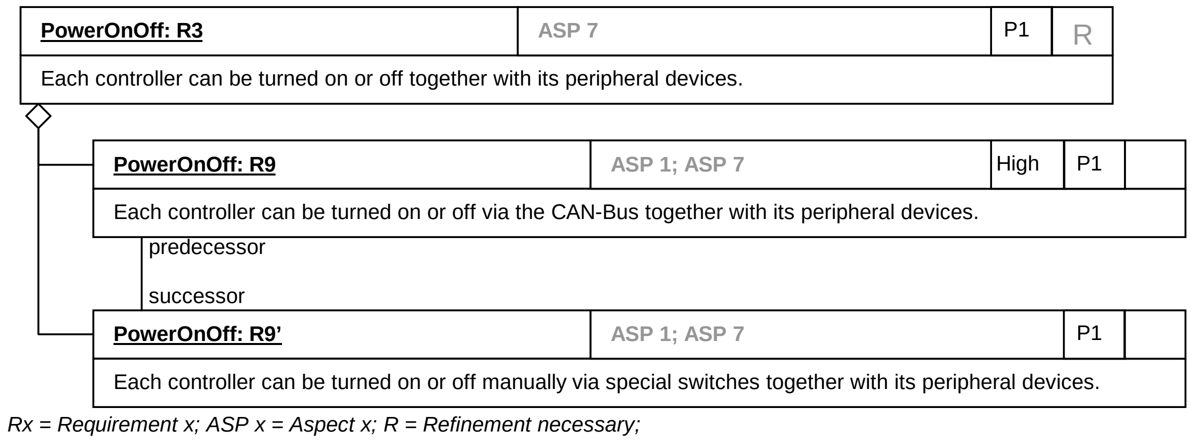

PowerOnOff: R3
ASP 7
P1
R
Each controller can be turned on or off together with its peripheral devices.
PowerOnOff: R9
ASP 1; ASP 7
High
P1
Each controller can be turned on or off via the CAN-Bus together with its peripheral devices.
predecessor
successor
PowerOnOff: R9’
ASP 1; ASP 7
P1
Each controller can be turned on or off manually via special switches together with its peripheral devices.
Rx = Requirement x; ASP x = Aspect x; R = Refinement necessary;

Figure 51: Refinements of Requirement R3

4.3.2.1.2 Aspect

The specification of large, complex systems often consists of a big amount of requirements with a lot of different dependencies between them. Working on such specification documents is highly time-consuming and error-prone. In order to cope with this problem different techniques for the reduction of complexity are provided by the presented model. One of them is the grouping technique “aspect”. The grouping of requirements is accomplished according to specific concerns of the developed system. For example the requirements presented in Figure 49 and Figure 51 are grouped by aspects ASP1-ASP10 from Figure 52. The requirements R2, R4-R7 are assigned to the aspect ASP2, which gathers requirements, describing the interface to the push buttons of the window-lift-system.

An aspect groups requirements according to their semantic dependencies. The assumption for all requirements belonging to one aspect is that they have semantic similarities according to the aggregating aspect. For example all requirements of aspect ASP2 describe the syntactical interface to the push buttons.

This semantic dependency can be expressed as a correlation function between a pair of requirements. Every aspect defines its specific correlation and gathers all requirements, which fulfil it. It is assumed, that the correlation is transitive, reflexive and symmetric. This assumption simplifies the real world correlations, which may be not transitive (or even anti-symmetric). This simplification permits the reduction of complexity and unification of dealing with big amounts of requirements. For example requirements on hardware and requirements on programming language could be gathered in one aspect called “operating system” if they correlate to requirements on operating system in some specific point of view. However, the same correlation function directly between a requirement on hardware and a requirement on programming language could be not applicable. Nevertheless the aspect “operating system” is useful, as it contains all requirements, which can be directly affected by some changes concerning the operating system in the specification document.

A further consequence of the above assumption is the possibility to reduce the search space for the impact of change. If a requirement changes, only those requirements can be directly affected which are in the aspects that aggregate the requirement. In order to provide this, all possible correlations to the other requirements should be considered in form of aspects and each requirement has to be a member of at least one aspect. These two constraints can be seen as a guideline for the usage of the presented conceptual model.

Plain partitioning of big sets of requirements using aspects can make resulting requirements groups quite large, so the aimed reduction of complexity and bordering of change impact can be only partially achieved in this case. In order to solve this problem, an observation can be exploited, that the requirements grouped by one aspect can usually be broken down to part-groups using the same technique.

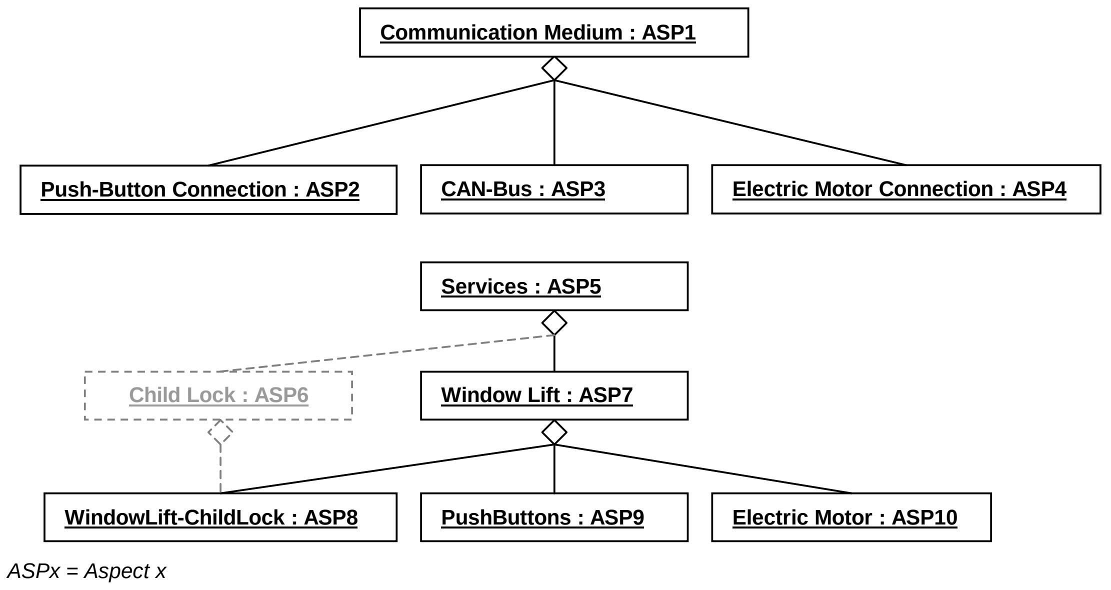


Figure 52: Aspect Hierarchy

The technique of building part-groups is aided by a hierarchy of aspects in the presented model. Figure 52 demonstrates the hierarchies of aspects in the window-lift example. An aspect can consist of several sub-aspects as well as of standalone requirements. This means, the sub-aspects partition their parent-aspect not completely. For example in the window-lift specification, R1 and R3 are standalone requirements of the aspects ASP7. They are not aggregated by any ASP7’s sub-aspects. Any pair of sub-aspects has also not to be distinct, e.g. one requirement can belong to several sub-aspects. For example the requirement R8 is aggregated by aspects ASP8 and ASP9 because it specifies the functionality of child lock as well as the functionality of push buttons. The hierarchical decomposition of aspects is a further mechanism for managing the complexity, provided by this conceptual model.

An aspect can be a part of several parent-aspects if all its members can be aggregated by the correlation functions of every parent. Suppose a new aspect ASP6 called “Child Lock” accrues to the aspect hierarchy from Figure 52 as a sub-aspect of ASP5. ASP6 groups the service of child lock, which has now to be realized in the presented fictional example of development process also. The aspect ASP8 gathers all requirements which specify the child-lock functions of the window-lift-system. So it is one of the sub-services of window-lift (ASP7) and could be aggregated by ASP6 also, since it is one of the sub-services of child lock too. The described technique allows the reuse of defined aspects and increases the expressiveness of the concept.

Using the concept of aspect hierarchy the efficiency of bordering the change impact can be increased noticeably. The change impact of a requirement can be assumed as bordered by the first ascendant in the aspect hierarchy, to which the requirement belongs to, whose children are not affected by this change. The same boundary can be assumed for the siblings of the changed requirement. For example the requirement R5 from the window-lift system changes to R5' as shown on Figure 33. R5 was aggregated by aspects ASP2, ASP9 and ASP10. Considering the grouping aspect of ASP2 nothing can be affected by this particular change of R2. However the members of ASP9 and ASP10 have to be examined, because the behaviour of the window-lift, concerning the push-buttons and motor control, has changed. The father-aspects of ASP9 and ASP10 have to be explored also (ASP7). Since none further requirements are affected, the examination of impact can be terminated.

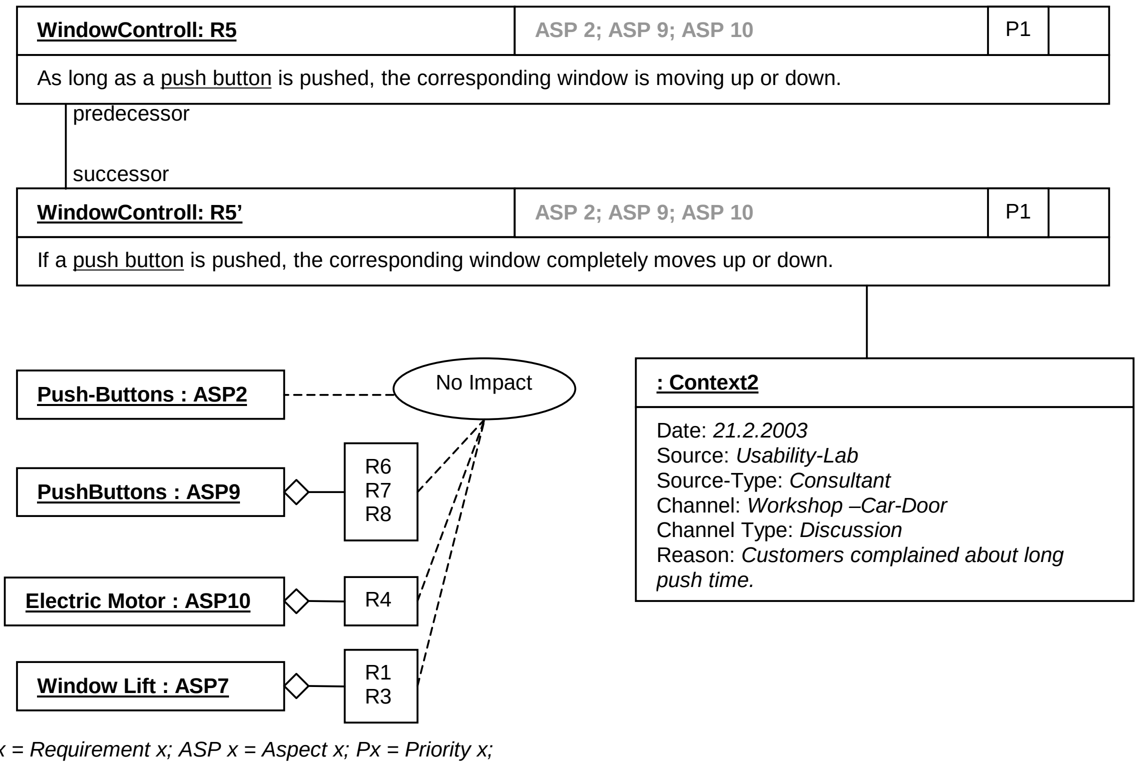
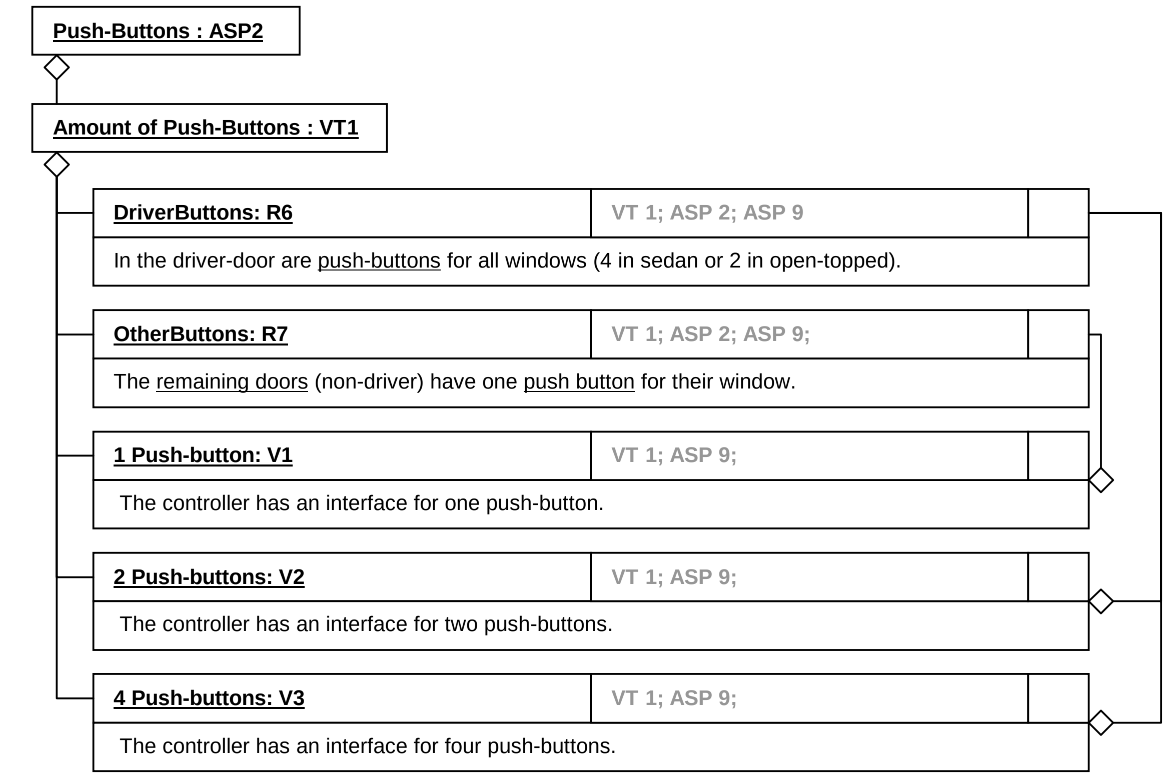

WindowControll: R5
ASP 2; ASP 9; ASP 10
P1
As long as a push button is pushed, the corresponding window is moving up or down.
predecessor
successor
WindowControll: R5'
ASP 2; ASP 9; ASP 10
P1
If a push button is pushed, the corresponding window completely moves up or down.
: Context2
Date: 21.2.2003
Source: Usability-Lab
Source-Type: Consultant
Channel: Workshop –Car-Door
Channel Type: Discussion
Reason: Customers complained about long push time.
Push-Buttons : ASP2
No Impact
PushButtons : ASP9
R6
R7
R8
Electric Motor : ASP10
R4
Window Lift : ASP7
R1
R3
Rx = Requirement x; ASP x = Aspect x; Px = Priority x;

Figure 33: Boundary of Change-Impact

The existence of several independent hierarchies is made possible by the model design (consider the two hierarchies on Figure 52). A requirement can belong to different independent aspect-hierarchies. The only limitation is that a requirement can not be a member in two aspects, which are standing in an ancestor/descendant-relationship to each other. This constraint assures the clarity and unambiguousness of aspect assignation.

4.3.2.1.3 Variation type

A specification document can describe a whole product family containing several products. Such documents often contain contradictory requirements, which specify differences between particular family members. These contradictions have to be handled in another way as the ones, described in the section about requirements. They don't make the whole model instance inconsistent, but have to co-exist in one specification at the same time. Such contradictions are captured using the concept of variations and variation types. An example of a variation type is given in the Figure 54. Its variations describe different amounts of push-buttons, which can be connected to a specified window-lift-system. Variations and variation types make it possible to describe the requirements of a whole product family in one instance of the model. So, this feature provides the support for Product Line Approach [CDK01].

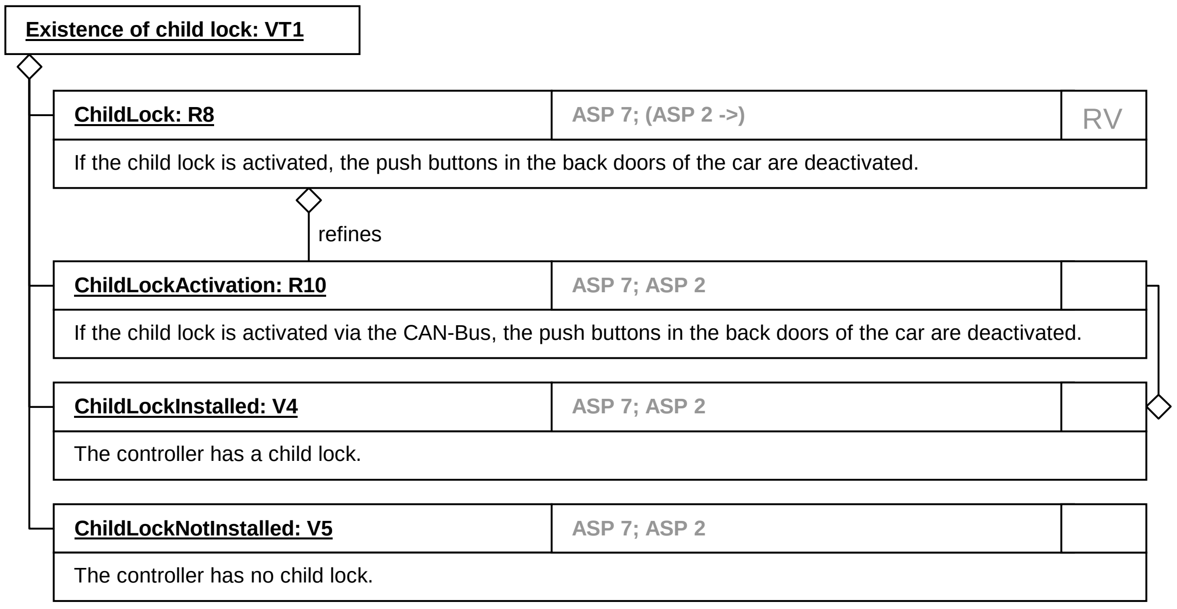

Push-Buttons : ASP2
Amount of Push-Buttons : VT1
DriverButtons: R6
VT 1; ASP 2; ASP 9
In the driver-door are push-buttons for all windows (4 in sedan or 2 in open-topped).
OtherButtons: R7
VT 1; ASP 2; ASP 9;
The remaining doors (non-driver) have one push button for their window.
1 Push-button: V1
VT 1; ASP 9;
The controller has an interface for one push-button.
2 Push-buttons: V2
VT 1; ASP 9;
The controller has an interface for two push-buttons.
4 Push-buttons: V3
VT 1; ASP 9;
The controller has an interface for four push-buttons.

Rx = Requirement x; ASP x = Aspect x; Vx = Variation x; VTx = Variation Type x

Figure 54: "Amount of Push-Buttons"-Variation

At some early stages of negotiation and analysis phases some contradictions cannot be resolved definitely. The strategy in such cases can be to track all alternatives simultaneously, until a decision can be made. For this purpose the variation and variation type mechanisms are also capable.

The artefact "VariationType" is derived from "Aspect". It aggregates sets of requirements, which contradict to each other according to the aspect the variation type was derived from. The union of these requirement sets is described by the aggregation, inherited from the artefact "Aspect", e.g. the sets partition the underlying aspect completely. Every set has exactly one representative requirement, called "variation". The contradiction between sets of requirements is defined as contradiction between variations. Variations are discussed in the following section 4.3.2.1.4 in detail.


Existence of child lock: VT1
ChildLock: R8
ASP 7; (ASP 2 ->)
RV
If the child lock is activated, the push buttons in the back doors of the car are deactivated.
refines
ChildLockActivation: R10
ASP 7; ASP 2
If the child lock is activated via the CAN-Bus, the push buttons in the back doors of the car are deactivated.
ChildLockInstalled: V4
ASP 7; ASP 2
The controller has a child lock.
ChildLockNotInstalled: V5
ASP 7; ASP 2
The controller has no child lock.

Rx = Requirement x; ASP x = Aspect x; R = Refinement necessary; V = Variants necessary; Vx = Variation x; VT x = Variation Type x

Figure 55: "Existence of Child Lock"-Variation

A contradiction, expressed by means of variation type, doesn't have any impact on the consistency of the model. That fact explains the difference between the variation types and the contradiction of requirements, introduced in the section 4.3.2.1.1. Contradicting requirements make the state of the model instance inconsistent, they express invalidities found during the requirements analysis, which have to be cleared and resolved in the future. Variation types represent volitional, explicitly planned contradictions.

Variation types can participate in the aspect hierarchy. For the objectives of change impact limitation as well as for the objective of grouping by semantically similarities no distinctions have to be made to the treatment of simple aspects. A variation type can include aspects and standalone requirements as well as other variation types as its children/descendants. This fact doesn't make any difference to the semantic meaning of a variation type because of the distinction between a variation type and its variations (see next section for details).

4.3.2.1.4 Variation

A variation is a special requirement, which contradicts with at least one further variation. A variation aggregates a set of requirements and describes therewith a partition of the variation type it belongs to, which was mentioned in the previous section. The union of requirement sets aggregated by all variations of a particular variation type is equivalent to the whole set of requirements, aggregated by this type. This means, that the partition is complete, however it is not necessarily distinct. Several variations are allowed to associate common requirements, which have in any case to be realized if one of the variations is chosen.

The grouping of requirements around a particular variation forbids the existence of any further contradictions between the group members according to the variation type and its correlation function. However this doesn't mean, that some of the group members are not allowed to contradict according to another correlation. This fact has to be considered in the model instance by defining the corresponding variation type.

As mentioned above a requirement can participate in several contradiction relationships to other requirements. In order to reflect this fact in the model, a variation can simultaneously belong to different variation types.

4.3.2.2 Design

In order to trace the realization of requirements during the whole development process, a connection to the neighbour discipline “design” is necessary. The model offers an interface, which consists of special artefacts and concepts for mapping requirements to the products of the design discipline. This interface serves for post- and pre-traceability purposes and allows consistency checks between specification and design documents.

This section introduces two artefacts, which are conceived as the mentioned interface to design. These artefacts can be refined for and adapted to specific needs of the application domain, as described in section 4.4.

4.3.2.2.1 Design Artefact

The conceptual model defines “DesignArtefact” as an abstract entity, which represents a work product of the design phase. The only demand concerning this entity is that it must associate all requirements it deals with. This under-specification leaves the possibility of adapting the model to a specific design paradigm open. The election of a design paradigm depends among other things on the specific application domain or characteristics of the developed product. Such paradigms identify different types of design products (e.g. module/interface, or component/connector/channel) as well as their specific interdependencies. In the presented conceptual model no specific roles or tasks are defined for design artefacts. Further on no interdependencies between design products (like needs-/offers-interfaces, or component-hierarchies), which belong to the same design document, are constituted by design artefact. The task of the concrete specification is delegated to the designer, who has to refine the abstract design artefact entity and to adapt it, to be applicable in the specific product domain. Section 4.4 gives a guideline and examples for the described activities.

Figure 56 offers a specific view on the model that outlines all artefacts and relationships, which are relevant for activities on the design artefact.

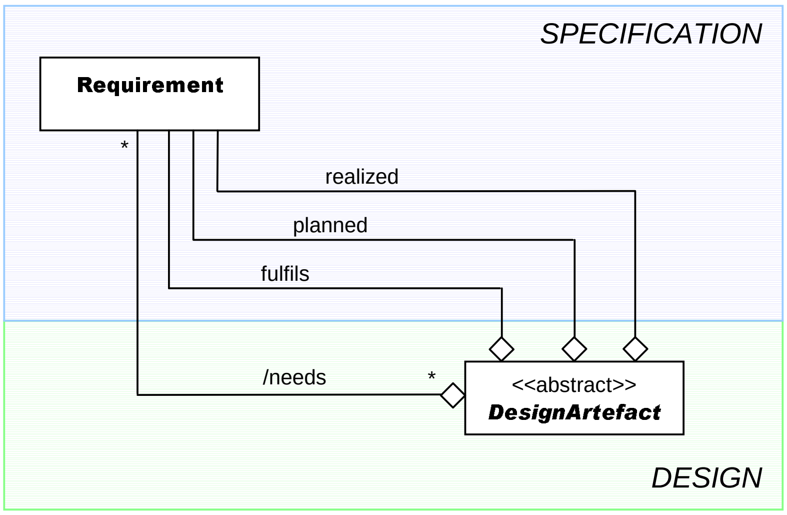


Figure 56: A View for Design Artefact

A design artefact can fulfil a set of requirements either by realizing them or by demanding their realization from other design artefacts. All these requirements are aggregated by the association “fulfils”. The set is partitioned by three further associations completely. These are:

- The association “realized” captures all requirements, realized by the design artefact itself.
- The association “planned” describes requirements, which have to be realized (but are not realized yet) by the artefact itself.

- The third association is called “needs”. It aggregates all requirements, which the design artefact demands from other design artefacts.

A requirement can either be realized or planned for a given design artefact, so the both associations are distinct. The derived association “needs” aggregates all requirements, which are not planned or realized. It is given implicitly and doesn’t appear in the class diagrams on Figure 48.

The associations map the requirements to the design artefacts and are very important for traceability and consistency purposes. The association “fulfils” defines all requirements the particular design artefact has to deal with. So the impact of a specification change on the design can be comprehended easier. The associations “realized” and “planned” document the actual realization status of a particular design artefact and are useful for tracking of the implementation status as well as for release planning (see section 4.3.2.3.1 for details). The association “needs” defines a consistency constraint for design: If a requirement is needed by some design artefact, it has to be realized (or planned for realization) by another design artefact. This elementary constraint for a correct design can be verified automatically, without need of human intervention.

The constraint described in the above paragraph can be stated formally for all design artefacts $d \in DA$ as $r \in d.needs \Rightarrow \exists \overline{d} \in DA : r \in \overline{d}.realized \cup \overline{d}.planned$, where the relation “needs” is defined as $d.needs = d.fullfils - (d.realized \cup d.planned)$. Note, that due to the definition of $d.needs$, no requirement can be planned/realized and needed by the same design artefact. This constraint can be seen as a criterion for the completeness of design, which is ready for implementation. However in order to keep the process flexible (see section 4.5), using the presented conceptual model, this constraint is not stated to be an invariant for the model.

Obvious consistency constraints, which are demanded to be invariants, are:

1. Realized and planned requirements build a subset of requirements, associated by “fulfils” relation: $d.realized \cup d.planned \subseteq d.fulfills$.
2. A requirement cannot be planned or realized at the same time for one design artefact $d.planned \cap d.realized = \varnothing$.

Another application area of the above associations could be the reuse of design artefacts in different design documents. Every design artefact is characterized by a set of requirements, it realizes as well as by the set of requirements it needs from others. These sets are one of the basic criteria which help the designer to make a decision, whether the reuse is possible or not.

4.3.2.2.2 Configuration

With the help of variations and variation types a whole product family can be described by one instance of the introduced model. During the design phase of a particular product, decisions have to be made, which of contradictory requirements (read: variations from section 4.3.2.1.4) will be realized in this concrete product. A set of all chosen variations identifies the developed product pretty good. This set is called *configuration*. For example one configuration of the window-lift system can realize a one-button interface variation from Figure 54, and a support for a child lock (see Figure 55).

Figure 57 shows the artefacts and associations which are relevant for definition and maintenance of configurations.

The artefact “Configuration” describes a particular product from a product family, acquired by the whole instance of the model. It aggregates a set of variations (see sections 4.3.2.1.4 and 4.3.2.1.3) for this purpose. Again, these variations represent alternatives chosen for realization of a concrete product. A requirement that belongs to one or many variations will not be realized in the configuration, if and only if none of these variations were chosen by the configuration. For the rest of requirements contained in the instance of presented conceptual model it is assumed, that they have to be implemented in the configuration, e.g. are all aggregated via the “fulfils”-association (see Figure 57).

A configuration is build of a set of design artefacts, which are described in the previous section. Design artefacts have to implement variations as well as other requirements which specify the developed product. Each requirement, chosen by a configuration, has to be realized by one (or several) design artefacts, associated with this configuration. This consistency constraint can be verified effectively using the presented model as a basis for the requirements management process. A design artefact can be aggregated by several configurations. This fact aids the reuse of design artefacts.

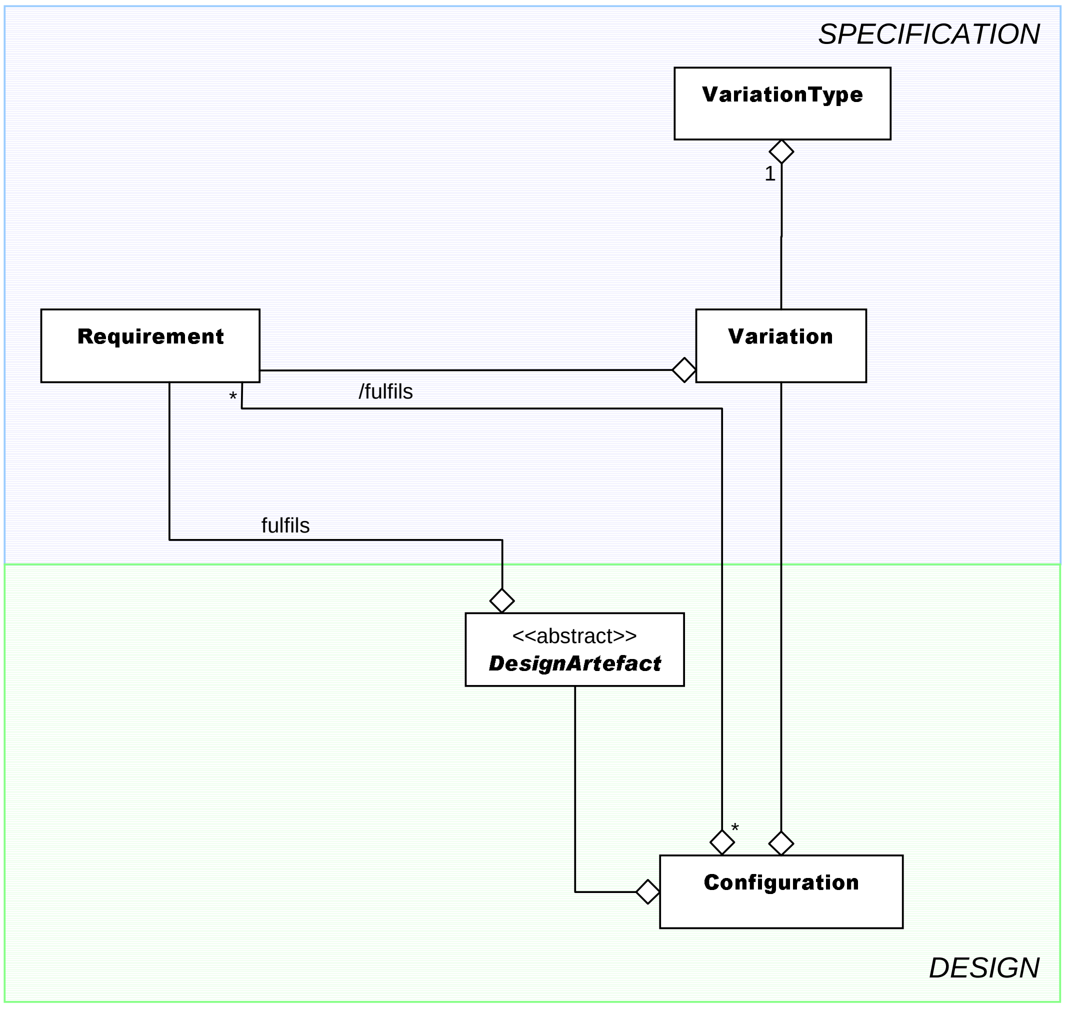


Figure 57: A View for Configuration Artefact

More formally, the relationships between any configuration $c \in CONF$, a set of associated variations VAR_c and the artefacts it aggregates are:

1. $c.fulfills = \{r \mid r \in REQ_c \vee (\forall vt \in VT : \forall v \in VAR_{vt} : \neg(r = v \vee r \in REQ_v))\}$, where
   - VT is the set of all variation types,
   - VAR_{vt} is the set of all variations, which belong to one variation type vt,
   - REQ_v is the set of all requirements aggregated by variation v and
   - $REQ_c = \bigcup_{v \in VAR_c} (REQ_v \cup \{v\})$ is the set of all requirements, aggregated by the variations of configuration c (VAR_c) plus the variations themselves.
2. $c.fulfills \supseteq \bigcup_{d \in DA_c} d.fulfills$, where DA_c is a set of all design artefacts, aggregated by the configuration c. This invariant allows for some requirements, which have to be implemented in the configuration not to be assigned to a design artefact. The aim of design process is to achieve the equality in the second constraint.

4.3.2.3 Implementation / Release Planning

Release planning constitutes the progression of the realization for a developed product. It helps to plan the implementation progress and to relate it to milestones and time constraints. In order to facilitate these activities, a set of realization states can be defined for every design artefact as well as for the whole configuration. These states are called *releases*. The planning of releases can occur before the implementation phase starts as well as during it.

4.3.2.3.1 Release

The development of a design artefact progresses continuously during the implementation phase. This progression is often planned as a sequence of implementation states, called releases. Releases, associated with one design artefact are totally ordered and have symbolic names (such as 0.2β, 1.3, 1.4.0.1RC3 and so on). Such a sequence of releases documents a planned development progress for a particular artefact. The short-term aim of implementation process is always to achieve the next planned realization state (release). As the realization state of a design artefact advances, the releases are achieved sequentially in total causal order.

The artefact “Release” represents exactly one release in the presented model. It is associated with one design artefact. Contrary every design artefact is associated with a totally ordered sequence of "Release" artefacts. The currently achieved release is associated to the corresponding design artefact directly, using the “implements subset”-link as shown in Figure 48 and Figure 58. The total causal order upon releases, which belong to the same design artefact, is designed as a (one-to-one) relationship “successor/predecessor”.

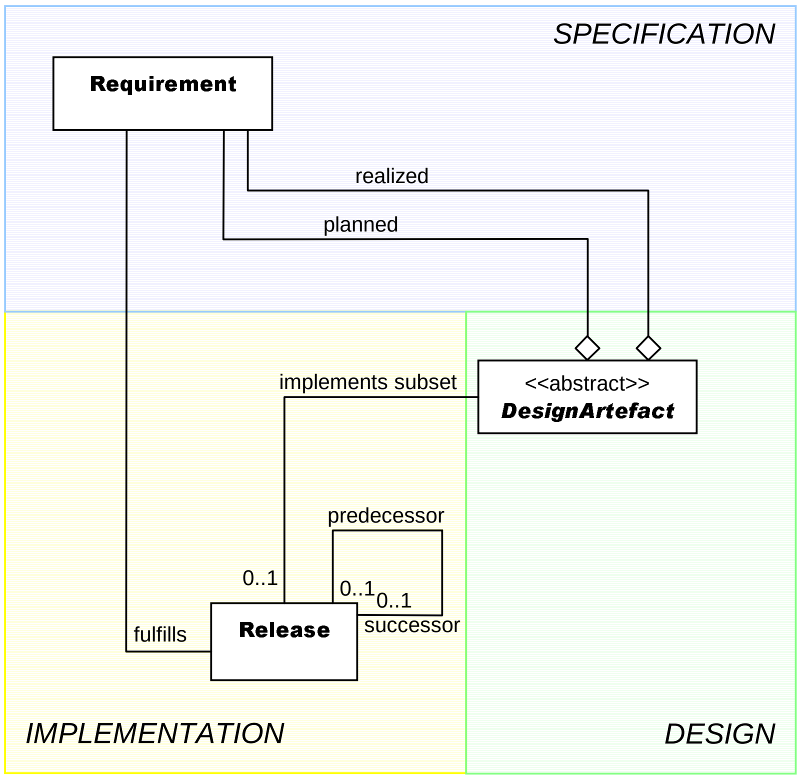


Figure 58: A View for Release Artefact

The realization state of a design artefact is defined upon the realization states of requirements associated with it. As described in section 4.3.2.2.1 a requirement can be in two states for a particular design artefact: planned and realized. The third state is called “needs”. It is implicitly expressed through the absence of a requirement in the “realized” or “planned” set aggregated by a particular design artefact. In the initial release all requirements are in the planned state. The following releases describe the sets of requirements, which have to be realized for each of them. These circumstances are expressed with the “fulfils” relation in the design of the conceptual model.

The change of a current release is allowed to take place only if the requirement set aggregated by the “realized”-association of the design artefact is equivalent to the state, described by the succeeding release. This event can be accomplished automatically, provided that the real development state is portrayed in the corresponding design artefact. Assuming the accuracy and validity of all previous release changes, it is sufficient to check the realization states of requirements aggregated by the following release and not by the current release and vice versa, in order to make decision, whether the release change may happen or not.

Any design artefact d and its release planning $\left(rel_0^d,\ldots,rel_{act}^d,\ldots,rel_{final}^d\right)$, where $rel_{act}^d = d.implements\_subset$ stands for a recently achieved release and rel_0^d, rel_{final}^d are start and end release, respectively, must stand at any time in the following relationships to each other:

1. For the next release rel_{act+1}^d to be achieved:
$$\left(d.realized - rel_{act}^d.fulfills \subseteq rel_{act+1}^d.fulfills - rel_{act}^d.fulfills\right)\wedge$$
$$\left(rel_{act}^d.fulfills - d.realized \subseteq rel_{act}^d.fulfills - rel_{act+1}^d.fulfills\right)$$
with $rel_0^d.fulfills = \varnothing$, where $A - B \overset{def}{=} \{a : a \in A \wedge a \notin B\}$.
2. The next release is achieved if $d.realized = rel_{act+1}^d.fulfills$.
3. For every future release $rel_{act}^d < rel_{future}^d \le rel_{final}^d$:
$rel_{future}^d.fulfills \subseteq d.planned \cup d.realized$.

The first consistency condition allows only changes on the actual realization status, which are recorded in the next release that has to be achieved. This condition guarantees a correspondence between the real implementation progress and its planning for every design artefact, e.g. no manipulations on the achieved release sequence are allowed and any not planned modifications on the current product release are forbidden. The second equation formalizes the condition of a release change, described some paragraphs above. It is a specialization of the first condition. The third condition postulates that the scope of every future release of the product has to be a subset of the product's planned realization status. This constraint provides consistency between the two options for planning: releases and realization status.

As an example for release planning consider the implementation priorities from Figure 49 on page 98. Suppose the realization of child lock support, which has a lower priority, is planned for release 2. The rest of the requirements has to be realized in the release 1 due to their higher implementation priority.

4.3.2.3.2 Configuration Release

The artefact "Configuration Release" defines the release planning for configurations. The dependencies between configuration releases and a configuration are the same as between releases and a design artefact. The interdependencies between single configuration releases as well as their aggregation of requirements are identical to the ones stated for releases in the previous section. Figure 59 shows a specific view on the conceptual model, which outlines all artefacts relevant for the configuration planning.

A configuration release aggregates a set of releases. This aggregation mirrors the relationship between a configuration and design artefacts, described in the section 4.3.2.2.2. It constitutes for each configuration release the releases of design artefacts, it consists of. So a configuration release can be expressed in terms of aggregated releases. A configuration release change is allowed only if all aggregated releases of design artefacts are already achieved. The management of this event can be automated using a simple algorithm.

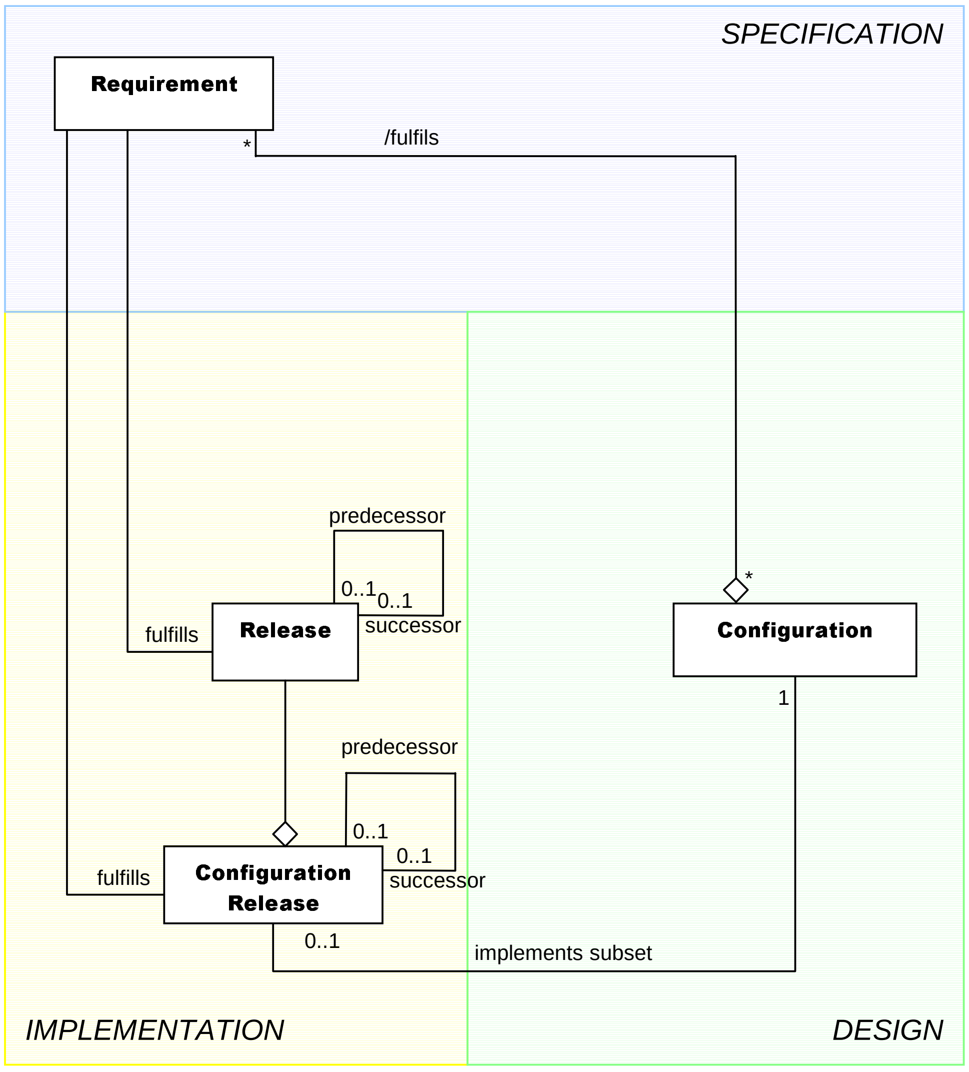


Figure 59: A View for Configuration Release Artefact

Concluding, there are two mechanisms which allow describing the release planning for configurations. The first is identical to release planning for design artefacts, e.g. it is based on delta-sets of requirements (see previous section for details). The second is based on composition of releases: A configuration release can be defined as an aggregation of design artefacts' releases, which the configuration consists of. On the one hand the two mechanisms induce some redundancy into the presented conceptual model, but on the other hand, they offer a great flexibility for the process, working on this model (see section 4.5 for details). The two redundant representations of configuration releases can be mapped to each other automatically, so (provided some tool-support exists) no inconsistencies can arise.

Formally for every configuration c and its release plan $\left(cr_0^c, \ldots, cr_{act}^c, \ldots, cr_{final}^c\right)$, where $cr_{act}^c = c.implements\_subset$ stands for a recently achieved configuration release and cr_0^c, cr_{final}^c are start and end configuration releases, respectively, the following relationships must be valid:

1. $cr_{final}^c.fulfills \subseteq c.fulfills$, where $cr_{final}^c.fulfills$ stands for a set of requirements, which has to be implemented in the end version of the product (configuration c) using the configuration release planning mechanism.
2. For every configuration release cr_i^c, with $0 \leq i \leq final$:
   $cr_i^c.fulfills \supseteq \bigcup_{rel_j^d \in REL_{cr}} rel_j^d.fulfills$, where REL_{cr} is a set of all design releases that belong to the configuration release cr_i^c and rel_j^d, stands for the set of all requirements, which have to be realized in the j-th release ($0 \leq j \leq final_d$) of the design artefact d according to the associated release planning. For the first release this inequality can be simplified to $cr_0^c = \varnothing$.

The first consistency condition states that the requirements planned for the final configuration release have to be a subset of requirements, aggregated by the corresponding configuration, since the goal of every development process is to realize all requirements standing in the specification. The second condition demands the set inclusion relationship for the both mechanisms of release planning at any time. The aim of the planning process is to achieve the equality in the second constraint. The set inclusion relationship is only desirable in the early phases of development, where the design has not been completely created yet.

4.3.3 Conceptual Model for Requirements of Real-Time Embedded Systems

In this section the conceptual model for requirements is further refined to fit the needs of the domain of real-time embedded systems. For this purpose, three approaches are pursued: Firstly, attributes of the artefact "Requirement" are specified. Secondly, a refinement of the design artefact is accomplished. And thirdly, some specific aspect hierarchies for the application domain are proposed.

4.3.3.1 Refinement of Requirement Artefact

In respect to the domain of real-time embedded systems, the artefact "requirement" is refined to an artefact "embedded requirement", and the following attributes are introduced:

- *environment* ("true" or "false"), to denote, if this requirements captures the functionality of the system to be developed, or if it describes the environment, in which the system has to operate
- *time-critical* ("true" or "false"), to signal, that a requirement describes a time property of the system
- *test-coverage,* describing the number of test-cases, that check the implementation of a requirement

4.3.3.2 Refinement of Design Artefact

Since the EMPRESS project focuses on the component-based software development, the artefact "DesignArtefact" is refined to an artefact "Component" (see Figure 60). The component inherits the properties of the design artefact (see section 4.3.2.2.1); especially it offers functionality, by either realizing that offered functionality, or by delegating parts of that functionality to other components.

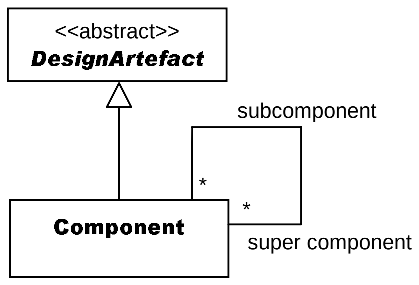


Figure 60: Specialization of DesignArtefact

Furthermore, every component can consist of several subcomponents. To reflect this hierarchy on components, the component artefacts define the "subcomponent"-relation. In order to provide reusability, one component can be included in several super-components.

4.3.3.3 Aspect Hierarchies

This section proposes types of aspects, which are suited to group requirements of real-time embedded systems. The structure of the aspects is shown in Figure 61

The functionality of real-time embedded systems can be often grouped by services which describe a certain part of the system behaviour. This grouping can be adopted by an instance of the presented model, by means of defining an aspect “service” as the root of a hierarchy. Figure 52 on page 101 demonstrates the service hierarchy for the window-lift-system example. The classification of requirements according their affiliation to a certain service should not be mistaken for deployment of functionality to components (or nodes) of a developed embedded system. This is a task of design, which can use the presented hierarchy implemental.

Another characteristic of real-time embedded systems is their ability to work with and to be integrated into other embedded systems. Thus the specification documents of embedded systems contain a lot of information about the communication with their environment. A grouping of the requirements concerning the communication can be performed by assigning requirements to communication partners and/or to the used communication media (channels).

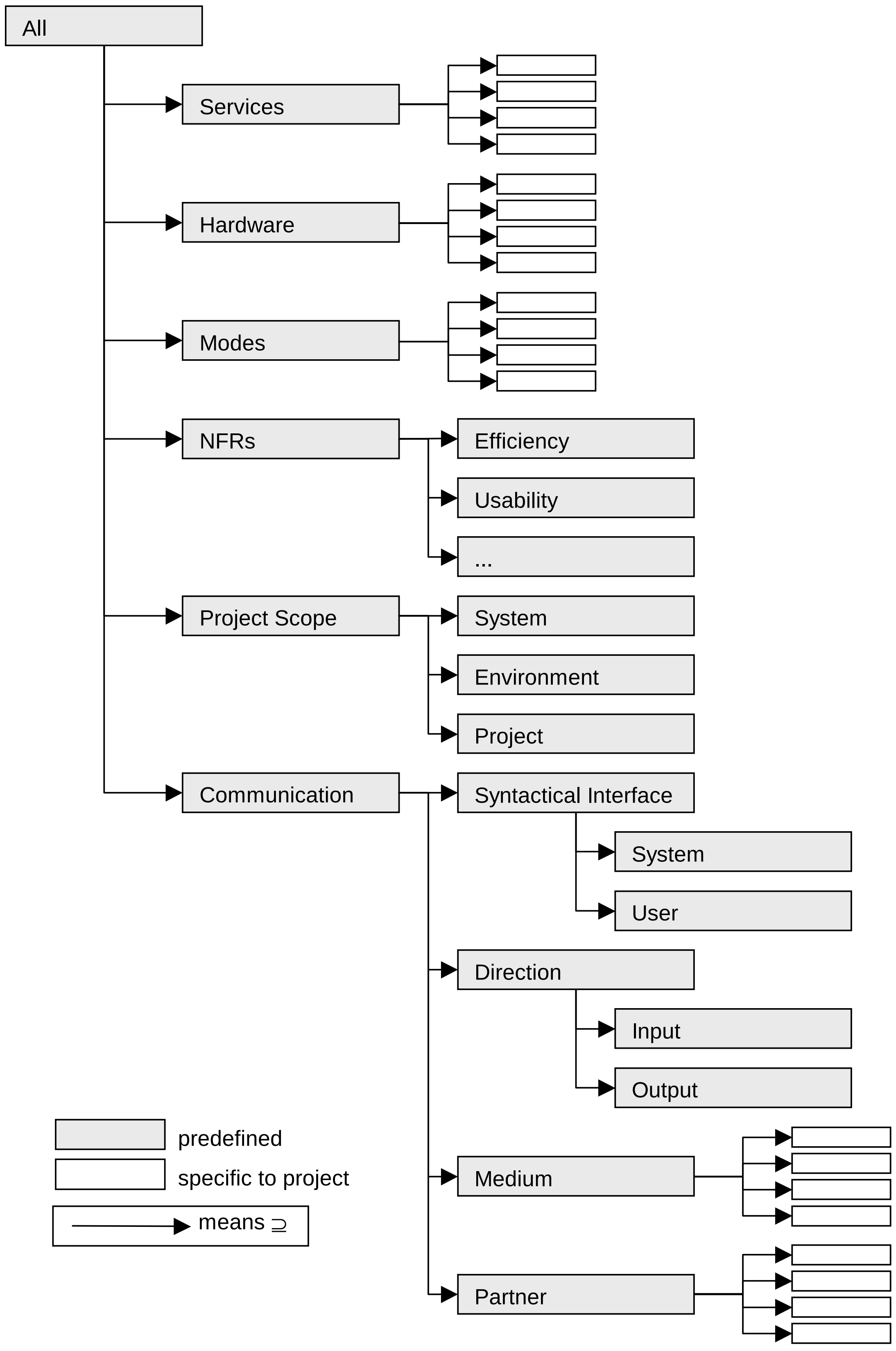


Figure 61: The Aspect Hierarchy Framework for Embedded Systems

The behaviour of an embedded system is often characterized by operational modes. Aspects can be used to group requirements belonging to one mode. For example the requirement R8 in Figure 49 could be assigned to an operational mode aspect “child lock on”.

The differentiation between the service hierarchy and the communication medium hierarchy is motivated due to the fact that several services can communicate via the same interface. Contrary one service can use several media. The hierarchy of interfaces serves as a good assistance for design activities. For example the service "ChildLock" and the service "PushButtons" represented by appropriate sub-aspects of service hierarchy on Figure 52 use both the CAN-Bus (see CAN-Bus aspect on the same figure). The control commands from child lock arrive at the window-lift system via the CAN-Bus. The commands from the push-buttons have to be delegated from the driver’s window-lift to the window-lift systems, they are addressed to. This happens through the CAN-Bus also. So the CAN-Bus is used by both services simultaneous.

While developing the whole embedded system (e.g. software and hardware) a further aspect hierarchy for partitioning the requirements can be proposed. It is the partitioning by hardware units. However, the EMPRESS-project deals with the software of real-time embedded systems only, so this hierarchy will not be presented in this contribution in detail.

4.4 Adaptation of the Model

The development of the conceptual model passed though several refinement steps (see Figure 26). The last refinement to the special application area has to be performed by the users of the model and can not be done by the TU München. In order to help the users in this step, the following subsections suggest some refinements for sample applications.

4.4.1 AutoFOCUS Design

The CASE-tool AutoFOCUS [AutoFOCUS] is specialized on modelling components of embedded systems with multiple diagrams. To demonstrate the idea of an adaptation of the model to this tool, a simplified description of the tool diagrams is given in the following.

Component-diagrams depict the functional units of the system. The interfaces between these components are designed with typed ports and channels. The elements mentioned so far describe the static structure of a system. The dynamic behaviour of the system can be characterized with Statecharts, which consist of states and transitions, and with Message Sequence Charts (MSC) which show exemplary execution traces.

The refinement of the model is in this case a specialization of the class “DesignArtefact”. The design elements of AutoFOCUS and their dependencies are shown in Figure 62.

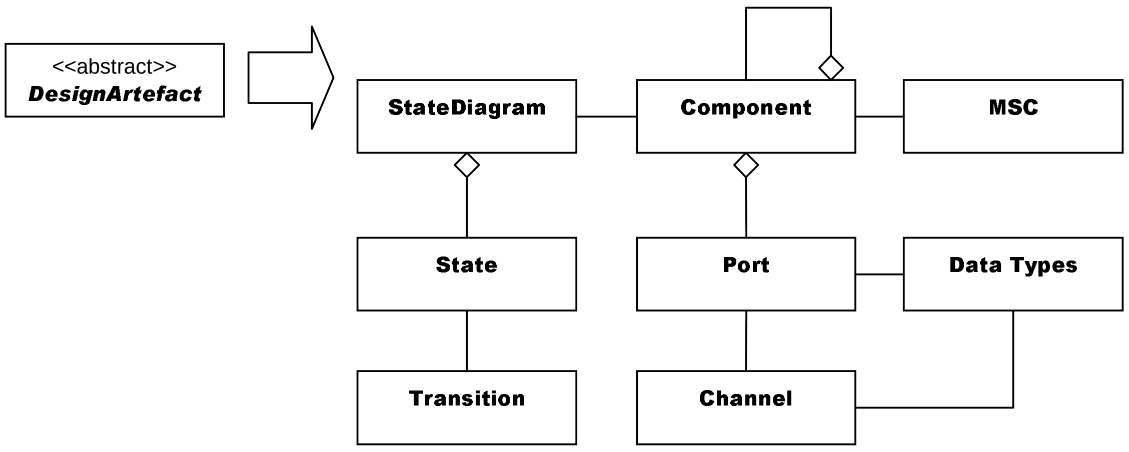


Figure 62: Specialization to AutoFOCUS

This refinement mirrors the concepts of the selected design paradigm. The same design structure is kept twice, for the assignment of requirements to design elements in an instance of the presented model and for the instantiation of the design elements themselves in an design document. Figure 63 illustrates the different usage of the design structure.

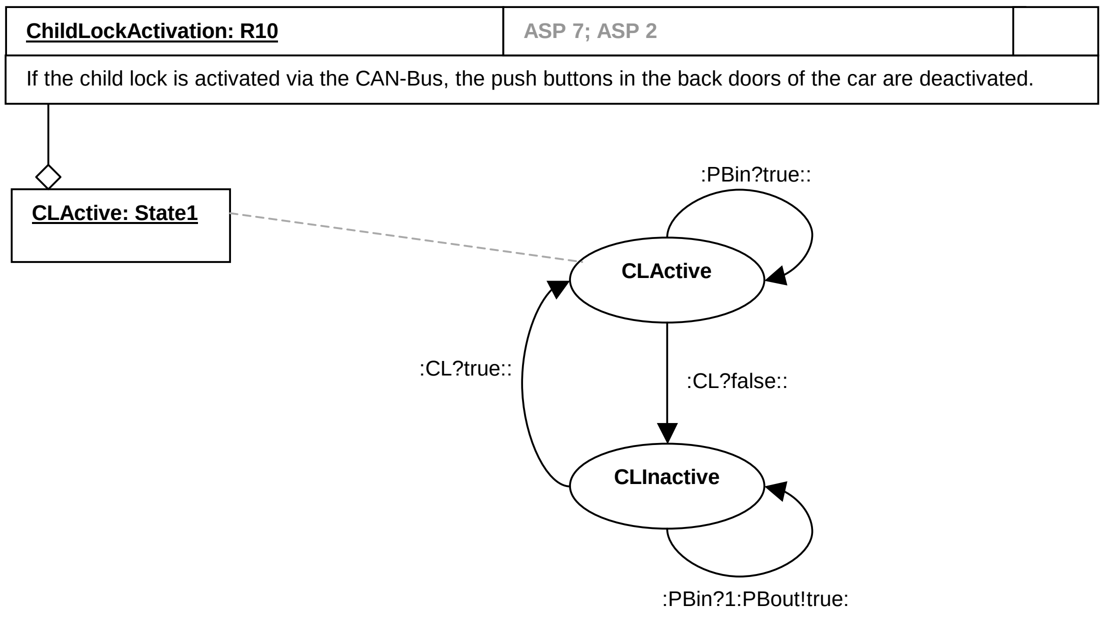


Figure 63: Associating Requirements to Design

In Figure 63 the state “CLActive” is assigned to the requirement R10. The figure also demonstrates the problem, how fine grained the structure of the design should be mapped to the conceptual model. If every design element is regarded, it could lead to some documentation overhead. For example, R8 from Figure 49 on page 98 would have to be associated with all transitions in the diagram. Otherwise the association to components only could lead to insufficient traceability. Here a decision of the designer has to be done, in order to get an acceptable interface to the design discipline.

4.4.2 Stand Alone Products

The conceptual model developed in section 4.3 contains elements to handle product lines. There may be several special application domains, in which product lines are not necessary. For these applications, the model has an overhead that can be removed.

The Product Line Approach is integrated orthogonal to the areas specification, design and implementation. It can be separated from the model by considering the three artefacts "VariationType", "Variation" and "Configuration" (see Figure 27). The construction of the conceptual model allows abandoning the PLA by removing these three classes and all associations connected to them. The "Configuration Release"-artefact is still remaining, though the name becomes senseless. The information stored in the objects of this class is assigned to the one configuration that is implicitly given by the specification. The resulting model (see Figure 64) is quite simplified.

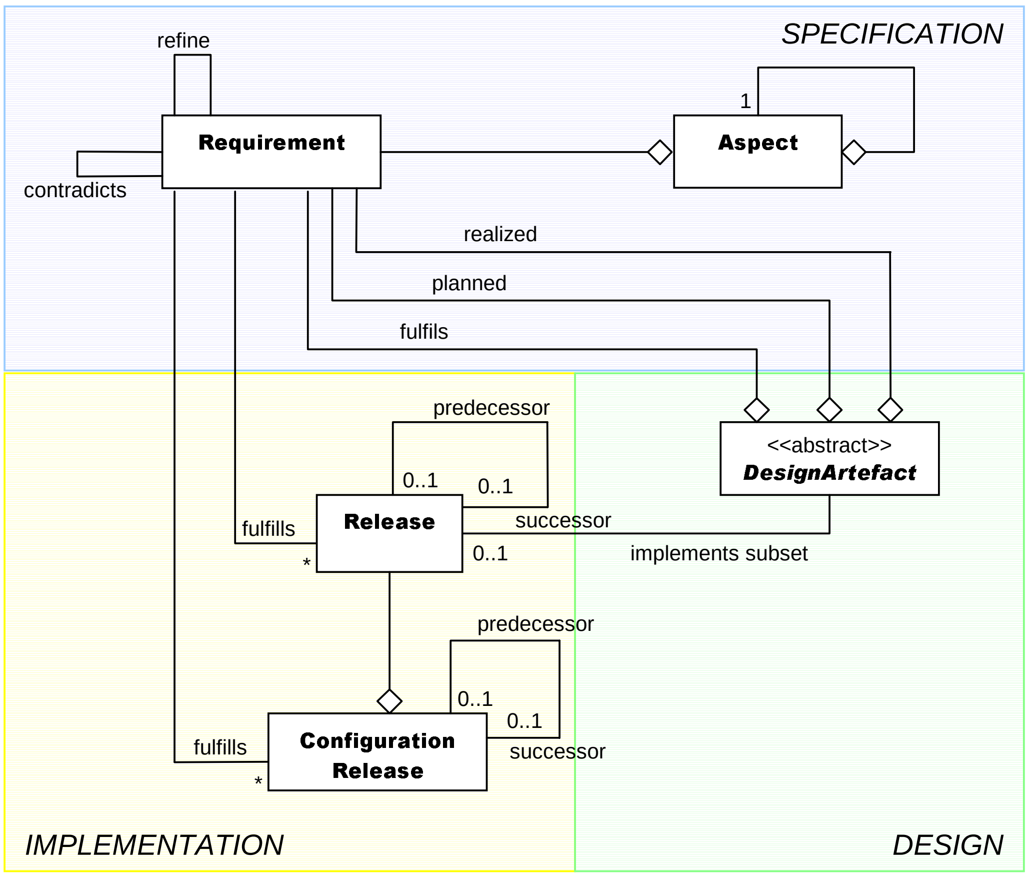


Figure 64: Tailoring to Stand Alone Products

4.4.3 Layers for a Refinement Hierarchy

The conceptual model in its basic version offers the association “refines” to document refinement relations between requirements. However, many approaches propose fixed layers for the refinement hierarchy. Karl. E Wiegers, for example, suggests grouping the requirements in a “vision and scope” document, a “use-case” document and a “software requirements” specification (see [Wie03]). The adaptation to this hierarchy can be done by tailoring the refinement relation and deriving three new artefacts including the relationships. Figure 65 shows the adapted model.

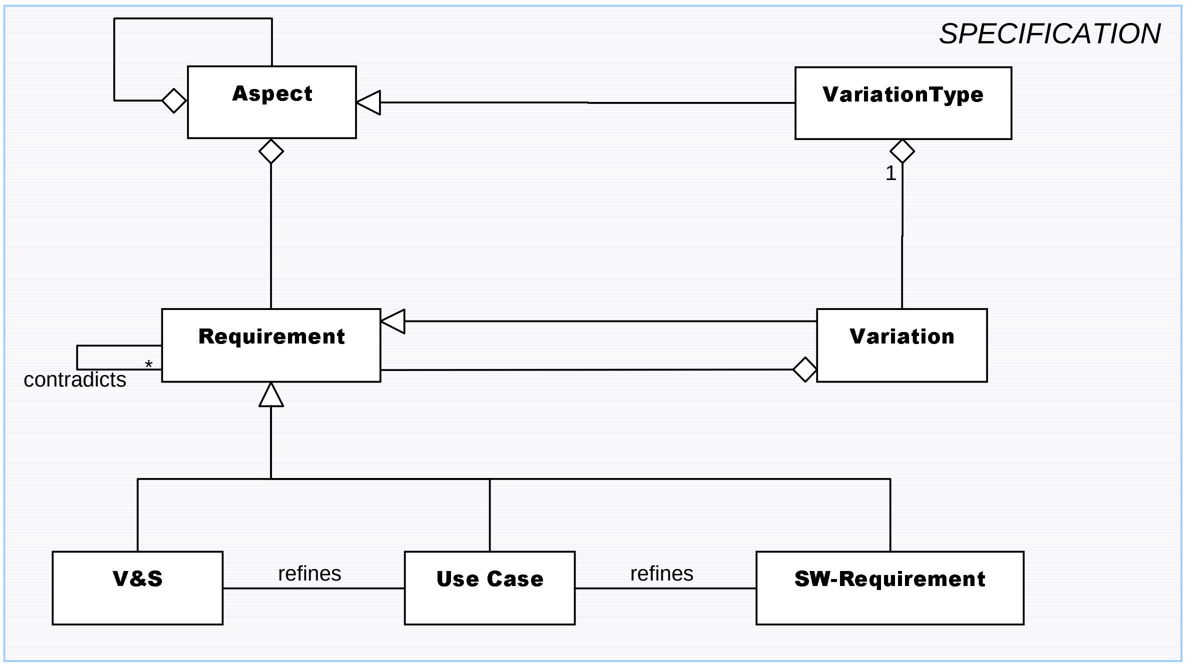


Figure 65: Integration of a Refinement Hierarchy

Another concept for a refinement hierarchy can be found in section 2.3 “Abstraction Layers … Automotive Domain” presented by DaimlerChrysler. It suggests to structure into vision and scope at top level, features and use cases in the middle and system requirements at the bottom level.

Similar to the design (see section 4.4.1) it is recommended to trade off, how fine grained the layers should be. On the one hand, the traceability can be improved by having many small groups. On the other hand the insertion of new requirements may become more difficult because some requirements can not be clearly assigned to one group. In this question, again the domain knowledge and the experience of the requirements analyst is the deciding factor.

4.5 Process Model

The last sections described the data structure in which the information for the specification is stored. This section describes a set of methods which together comprise a part of the process for requirements engineering. Therefore at first the methods are mapped to the activities of the EMPRESS-process. Then a notation for the methods is introduced. Based on this notation, the methods are presented in the remainder of this section.

4.5.1 Overview

The methods in this section are not covering the whole requirements engineering process. They are only related to the handling of the data structure. For example, methods concerning elicitation are not mentioned in here. The intention of the following sections is to give an overview of the handling of the model.

Process models often formulate a strict order of activities or at least a causal dependency between activities. The EMPRESS-process decouples activities from phases and therewith from a strict order. Though the activities in read world projects are in a causal dependency it is not denoted explicitly. It can be assumed, that an actor knows, when an activity makes sense. Especially in evolutional systems, the application of an activity is dependent on many factors and an execution order is very difficult to prescribe.

4.5.2 Integration into the EMPRESS-Process

The applicability of the conceptual model correlates with the ability to integrate it into the applied process models. Thus the conceptual model has to be integrated into the EMPRESS process. Within that process, the methods of this chapter can be mainly related to the requirements engineering and requirements management disciplines. Thereby the elicitation and negotiation of requirements are not covered. These activities are for example treated in section 6.1 (Hood). Table 4-1 introduces the methods and maps them to the subdisciplines of the EMPRESS-process.

	Subdiscipline			
Nr.	***Method***	Analysis	Documentation	Tracing
1.1	Adaptation to Design Paradigm	X	X	
1.2	Definition of Refinement Hierarchy	X	X	
1.3	Definition of Aspect Hierarchies	X	X	
2.1	Filtering of Requirements	X	X	
2.2	Integration of Requirements into the Aspect Hierarchy	X	X	X
2.3	Documentation of Variations	X	X	X
2.4	Extracting Specification Information	X	X	X
3.1	Creation of Design Artefact		X	X
3.2	Creation of Configuration		X	X
3.3	Mapping of Configuration to Design Artefacts		X	X
4.1	Creation of Release Plan for Design Artefact		X	X
4.2	Creation of Configuration Release Plan for Configuration		X	X

Table 4-1: Mapping of activities to the EMPRESS-process

4.5.3 Model and Notation for the Methods in the Process

The structure of a method is shown in Figure 66. A method is performed by a group of persons with a role (such as Designer). When describing a method it is also necessary to declare, which roles are allowed to perform it. A detailed description of the roles can be found in section 4.5.4. The inputs and results of the methods are development products. This means the results can be changed input products or newly created ones.

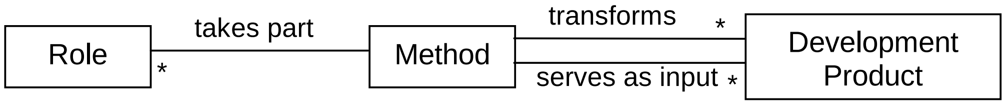


Figure 66: Structure for Methods

The description of the methods is done with tables (see Figure 67). Those tables contain the informal description of the method as well as affected roles and development products.

Nr.	Method Name
Input	*<<List of development products>>*
Precondition	*<<Conditions that have to be true before the method can be applied>>*
Description	*<<Detailed description of the method>>*
Roles	*<<List of involved roles>>*
Post-condition	*<<Conditions that have to be true after the method was applied>>*
Result	*<<List of development products>>*

Figure 67: Template for a Method

4.5.4 Roles

In most projects responsibilities for different activities are defined. This section suggests three roles, which are authorized to manipulate data in the instances of the conceptual model.

- **Requirements analyst**
  The requirements analyst is responsible for creating the specification of the system. His activities are placed in the requirements engineering discipline and he has to stand in a close contact to customers or people eliciting requirements.
- **Designer**
  In the requirements management discipline, the role “designer” is responsible for traceability to design documents in the conceptual model.
- **Release planner**
  The implementation of the design has to be planned by this role.

4.5.5 Setup and Refinement of the Conceptual Model

1.1	Adaptation to Design Paradigm
Input	-
Description	The abstract “DesignArtefact” is refined to a specific structure that represents the design paradigm of the system to be developed. A design paradigm could be for example object orientation or component architecture. The designer first determines, how fine grained the design paradigm should be represented in the model. Section 4.4.1 offers for example a detailed refinement to a AutoFOCUS paradigm. The designer can also add associations between the new artefacts for tracing purposes.
Roles	Designer
Result	Refined “DesignArtefact”

1.2	Definition of Refinement Hierarchy
Input	-
Description	Many approaches suggest three layer concepts (e.g. need, feature, system requirement) as presented in section 4.4.3. The original model offers a refinement hierarchy without layers. The artefact "Requirement" and its "refine"-association are adapted to the requirements engineering process of the project.
Roles	Requirements analyst
Result	Refined "Requirement"

1.3	Definition of Aspect Hierarchies
Input	-
Description	The aspect structure is created by instantiating the artefact "Aspect". Using the domain knowledge, helpful hierarchies are identified and manifested in the data structure (see for example section 4.3.3.3). A structure for requirements should be created, before they are inserted into the specification.
Roles	Requirements analyst
Result	Model with an instantiated aspect structure

4.5.6 Documenting and Structuring Methods

2.1	Filtering of Requirements
Input	Requirements
Description	The elicitation phase is a continuous process. For baselining (see [Doors]), the specification status is frozen and used as a base for the design activities. The requirements analyst determines the set of requirements that is visible for the design by marking each requirement with a status "approved" (see section 4.3.2.1.1).
Roles	Requirements analyst
Result	Requirements with status "approved"

2.2	Integration of Requirements into the Aspect Hierarchy
Input	• Requirements • Aspect Hierarchies
Description	Requirements that are added to the data structure are inserted in the aspect hierarchies in order to keep the specification structured. Within an aspect hierarchy a requirement can either be assigned to existing aspects, or a new aspect can be created, that opens a new requirement group.
Roles	Requirements analyst
Result	Links between requirements and aspect hierarchies

2.3	**Documentation of Variations**
Input	Aspect with contradicting requirements
Description	The contradiction of requirements can be resolved by creating a product line. In the model it is expressed with variation types and variations. The creation of a variation type includes following actions: • Creating variation type • Creating variations • Associating the variations to the variation type • Assigning requirements to the variations
Roles	Requirements analyst
Result	Variation type with variations

2.4	**Extracting Specification Information**
Input	All requirements, refinement hierarchy, aspect hierarchy
Description	Stakeholders often want certain information of the instance of the conceptual model. This information can be supported by generating specific views on the model (e.g. to support consistency analysis). The structuring and filtering mechanisms for theses views are the refinement and aspect hierarchies. The refinement hierarchy can be used to get the requirements on different levels of detail. This is, for example, useful for overview purposes and for understanding the motivation of low level requirements. The aspect hierarchy is used to extract requirements concerning certain aspects like services and modes. These aspects can be intersected until the set of requirements is reduced to an adequate size.
Roles	Requirements analyst, designer
Result	Relevant set of requirements

4.5.7 Tracing to Design

3.1	**Creation of Design Artefact**
Input	A set of approved requirements
Precondition	• The DesignArtefact is refined to the design paradigm. • There are no contradictory requirements, which are not captured as variations (see sections 4.3.2.1.1, 4.3.2.1.3 and 4.3.2.1.4 for details).
Description	The designer creates a specific instance of “DesignArtefact”. He links it to other instances of “DesignArtefact” using appropriate associations. The following relationships are arranged afterwards: • The “fulfils” relation is established between this design artefact and all requirements, which it has to fulfil by realizing them itself or demanding their realization from other artefacts. • The relation “planned” aggregates all requirements the design artefact has to realize itself.
Roles	Designer
Post-condition	The relation “realized” is empty
Result	Design artefact linked to requirements

3.2	**Creation of Configuration**
Input	• Variations • Approved requirements
Precondition	There are no contradictory requirements, which are not captured as variations (see sections 4.3.2.1.1, 4.3.2.1.3 and 4.3.2.1.4 for details).
Description	Designer creates a configuration and chooses the set of variations, the configuration has to realize. Links between the configuration and selected variations are established. Based on this selection a set of requirements is assigned to the configuration according to the rules described in section 4.3.2.2.2, constraint 1. This set is aggregated by the “fulfils” relation of created configuration.
Roles	Designer
Result	Configuration linked to variations and requirements

3.3	Mapping of Configuration to Design Artefacts
Input	• Configuration • A set of design artefacts
Description	Designer associates a set of design artefacts with the configuration.
Roles	Designer
Post-condition	The sets of requirements captured by the “fulfils”-association of the configuration has to be a superset of the union of the requirements sets aggregated by “fulfils” relations of affected design artefacts, as pointed out in section 4.3.2.2.2, constraint 2.
Result	Links between design artefacts and configuration

4.5.8 Tracing to Implementation

4.1	Creation of Release Plan for Design Artefact
Input	Design artefact with associated requirements
Description	The release planner creates an ordered sequence of releases for one design artefact. The sequence has the initial release with an empty ”fulfills” association as its starting point. The initial release is linked to design artefact using the “implements subset” relation. Every following release is described through a set of requirements it must fulfil and the difference to the previous release can be obtained by comparing it to the previous one.
Roles	Release planner
Post-condition	• As it was stated in section 4.3.2.3.1, constraint 2, the set of requirements, planned using releases and the realization status captured by design artefact, using “fulfils”, “realized” and “planned” relations must be kept in a certain relationship to each other. • If the design artefact is assigned to one (or several) configurations, the release planning must correspond to the configuration release planning, as described in section 4.3.2.3.2, constraint 2.
Result	• Linked list of releases for design artefact • Associations between releases and requirements

4.2	Creation of Configuration Release Plan for Configuration
Input	• Configuration • Associated requirements • Associated design artefacts
Description	The release planner creates an ordered sequence of configuration releases for one configuration. The sequence has the initial release with an empty “fulfils” association as its starting point. The release is linked to configuration using the “implements subset” relation. Every following configuration release is described through a set of requirements to be realized.
Roles	Release planner
Post-condition	• Configuration release plans and the requirements aggregated by “fulfils” relation of corresponding configuration must stand in a certain relationship to each other, which is described by constraint 1 from section 4.3.2.3.2. • The set of requirements planned using configuration releases and the union of releases of all design artefacts, which belong to the configuration, have to stand in a certain relationship to each other, which is described in section 4.3.2.3.2, constraint 2.
Result	• Ordered sequence of configuration releases • Associations between configuration releases and requirements

4.6 Conclusion

The approach for requirements engineering and management of evolving real-time embedded systems, as presented by TU München in this chapter, describes a central data structure for requirements. The three most important features of that model are:

- Firstly, the model serves the requirements engineering discipline by offering a data structure to adequately capture, structure and analyze requirements (with data elements like “Requirement”, “Context”, “Aspect”) as well as limiting the impact of changes. The data structure is suited for describing complete product families.
- Secondly, the model also supports the requirements management discipline by serving as a connection to the other disciplines, which can supply tracking information by establishing a relationship between their work products and the requirements (with elements like “design artefact”, “release”).
- Thirdly, refinement techniques are introduced, which allow an adaptation of the presented model to specific domains. The use of those techniques is demonstrated in the first parts of the chapter by using them in order to develop a specific data structure for real-time embedded systems.

Using those refinement techniques, the presented model can be adapted to other specific domains.

An activity-based requirements engineering process, presented in the last part of this chapter, describes the application of the presented model in the development of real-time embedded systems. The process consists of a number of methods, which can be applied to an instance of the conceptual model.

5 Elicitation and Analysis Methods and Models

5.1 A method to derive categorized requirements from a set of initial customer requirements documents

5.1.1 Introduction

The present study relates to the following research questions :

- Requirements classification, from unstructured input data
- Requirements elicitation, in conjunction with input requirements structuring

Both activities, namely elicitation and classification, are principal elements of the EMPRESS Requirements Engineering (RE) process. They represent first steps in the chain of related activities. The present study performed by HOOD covers sub activities in these fields, proposes related methods and techniques for the solution of related problems, and shows which parts could be covered with the support of appropriate tools. The mapping of these methods and activities to the EMPRESS RE sub discipline as defined in the introduction (ch.1) of the present report is shown below (Table 5-1):

RE Activity	RE Sub-Activity	RE Method covered by chapter 6.1
Elicitation	Define Scope	N/a
	Collection	Identify requirements and non-requirements from the information included in requirements input documents from different stakeholders
	Common Understanding	N/a
	Negotiation	N/a
Analysis	Classification	Structure the identified requirements according to the categories needed for the project information model
	Verification and Validation	N/a
	Identification of discrepancies	N/a
Documentation	Visualisation of contents	N/a
	Structuring of contents	N/a

Table 5-1 Mapping of chapter 6.1 methods to the RE activities substructure

At the beginning of any development project there is usually a lot of source information available from the customer. Unfortunately this source information is often not in a usable format and can take many forms in structure and in content. Stakeholders providing requirements often give functional requirements mixed with constraints and implementation details. Requirements can therefore not be identified clearly and not properly be traced. The stakeholders often make undocumented assumptions about the way they work and about their own particular domain; the requirements engineers are not usually domain experts and therefore are often not in a position to properly identify the context and the information contents intended by the author of the source information.

The purpose of the present study is to provide the requirements engineer with a process including methods, tools and notation in order to allow him to overcome the above-mentioned difficulties. This concept will support a logical approach to the problem of elicitation and furthermore classification of input requirements. The process is intended to de-risk and optimise the activity significantly, which supports the major business objectives of higher efficiency and lower cost of RM&E efforts. Depending on the types of input documents, the used grammatical style and vocabulary, the method may need to be adapted. However, the work performed shows that a methodical approach to the problem solution is possible.

Due to the fact that EMPRESS is concerned with real-time embedded software systems, the present study would have to concentrate on software aspects. However, since the reference requirements document is not explicitly addressing software, but mainly hardware, this could not really happen. On the other hand the result can be extended to hardware systems as well.

The aim of the study is to identify the first steps to do after receipt of the customer’s input requirements documents, in order to well prepare the contractor for the consultations with the customer on the common understanding of the requirements, and also to enable the contractor’s team to go into the details of technical assessment of the requirements. However it is pointed out that the study is a first attempt to get some knowledge and experience in this field, and to identify further research tasks still to be done.

The activity of “digesting” requirements inputs from the customer, as described in this chapter, embraces two main topics, namely the elicitation and the categorization / classification of requirements. Due to the fact that within the elaborated rules and process both topics are mixed thoroughly, it is not meaningful to split the study and this chapter into these two main areas. The part of elicitation is embracing all activities to separate requirements information in the input documents from non-requirements information. The classification part is concerned with the activities necessary to set attributes for each requirement, according to their related level of abstraction.

It may be of interest that those projects, which base on a system already developed and existing in hardware and software do not always have related requirements documents. In this case the requirement analysts could gain information from existing operational manuals and other descriptive technical documentation. For these cases the analysis of texts in order to find requirements might be of interest. This however is a specific topic, which must be analysed separately, since the rules found here may not be applicable for such cases.

5.1.2 Glossary of Terms

The following table provides the most important definitions of terms used in chapter 5.1. These terms are not part of the EMPRESS Glossary as laid down in Deliverable [D1.1] part 5, since they are specific to the current chapter 5.1.

Table 5-2 Glossary of terms for chapter 6.1

Term	Definition	Abbrev.
Contractor	Is the role, which gets the input documentation and has to develop the product, which meets all the requirements from the customers.	

Customer	Is the role, which is providing the input documentation as a contractual part of the development project. It authorizes requirements acceptable by him from all known stakeholders.	
Design Requirement	Requirements, which imply a certain technical solution and thus may limit the optimisation of the design.	**DR**
Information Element	An element of the input requirements documents, consisting of sentences, headers, bulleted lines, table cells etc.	
Input Documentation	A set of documents which includes in the eyes of the customer the contractual relevant input requirements, input requirements document, glossary, vision documents etc. As such it embraces the requirements from different stakeholders.	
Input Requirement	An input requirement represents the needs of the customer or of another stakeholder. It may consist of operational and functional rqmts., contractual rqmts., management rqmts., design rqmts., cost requirements, quality rqmts. etc. Input requirements are part of the Input Documentation defined above.	**InR**
Interface Requirement	An interface requirement provides technical information on the interface between the system developed and adjacent systems (of the same level, as well as of higher and lower levels)	**IR**
Product Tree	Hierarchical structure of a system, embracing all hardware and software parts of all abstraction levels from subsystems down to components.	
Stakeholder	A stakeholder is a person, which is effected by the system to develop or is directly related to or involved in the development or use of the system.	
System Requirement	A requirement out of the User Requirements, which is necessary to develop the system and verify it consistently. These requirements contain already a certain level of design information and background, namely from the highest-level architectural design.	**SR**
User Requirement	All Customer Requirements necessary to meet the technical goals and the set of requirements laid down in the Input Requirements. These requirements should not contain any design requirements. These will be covered in the System and Subsystem Requirements Documents.	**UR**

5.1.3 Input Requirements and the Problem

5.1.3.1 Overview

The overall view of what is to be achieved via the present study is explained in Figure 5-1. It shows that from a set of inhomogeneous information, which is considered to be requirements information, it is tried to achieve an information structuring. The requirements of the Input Documentation (document names are written with capital letters for section 5.1) are hidden among other information, and are not always easily visible due to writing style characteristics. However, the authors of these documents have certainly applied rules implicitly, which must be found out in order to solve the problem. At the end of the categorization effort there will be two sets of data, namely a structured one which is further used for the requirements engineering of the system to be developed, the other would not be used since not relevant. The following paragraphs provide some more detailed information on this general approach.

The classification for requirements and related documents used in this study has been selected following the nomenclature and definitions of chapter 2.3.2, which relates to the automotive domain. This is correct since the document used in the study as a basis for rules definition is from this technical domain. The classification has not used the level of business requirements, however has been extended to include as well subsystem, interface, design (= non functional requirements), and “other” requirements. The latter have been defined only for the case that due to a badly formulated requirement the categorization was not immediately evident, and may not be necessary for real requirement categorization efforts.

As a starting point a simplified data model of the requirements engineering and its adjacent activity fields is presented in Figure 5-2. This diagram shows the relationships between the various documents in a project and the abstraction layers they relate to. In practice input requirements contain elements of *user requirements, system requirements* and perhaps *subsystem* and *equipment requirements*, *architectural* or even *detailed design*, which is not the ideal case (for the definitions of these types of artefacts refer to para. 5.1.5.2.3). The task of the process described in this report is to support the decomposition of the input information into relevant functional and non-functional requirements and other, not directly used information. The latter is for example a comment, an explanation or context for a requirement and needs not to be transformed into a requirement.

It has to be pointed out that, although the notion *user requirements* implies that the user of the system under development should provide this document, this document is in most real cases not written by the customer. However, he will provide the basis for this in writing the *input requirements*. On this basis, the contractor will establish the formal User Requirements Document (URD), structured and formulated in an optimised way, and gathering all relevant and acceptable sets of requirements originated by different stakeholders. It is the intention of the presented process that this gap between the *input requirements* and the URD be closed by a systematic approach.

The initial tasks to be fulfilled within the overall procedure of requirements establishment are shown in Figure 5-3 and detail the activity box “Input Requirements” of Figure 5-2. The chain of activities is described as follows:

After an elicitation and gathering effort a Requirement Document or a set of various requirements documents from various stakeholders is written or provided by the customer and issued to the contractor. It is then the task of the contractor to identify which elements of the text contains real requirements, and which parts are of descriptive nature and provide context and explanatory information. Only after this task is performed, the requirements should be analysed by the contractor with respect to its technical content, its feasibility, whether they are unique and definite etc. This part of the requirements engineering will in general yield a set of questions produced by the contractor, necessary to be discussed with the customer. It will aim for eliminating all uncertainties of understanding and possibly will as well identify requirements omitted by the customer. The Input Requirements Document(s) should then be updated accordingly. This document issue will be the basis for writing the User Requirements Document, the System and Subsystem Specifications for the development of the system.

The set of input requirements documents must undergo configuration management starting from the first delivery of this documentation to the contractor, identifying baselines and versions. This is necessary to keep a detailed trace to all changes, which may occur starting from first delivery, and to make sure that any unintended changes by any persons involved are under control.

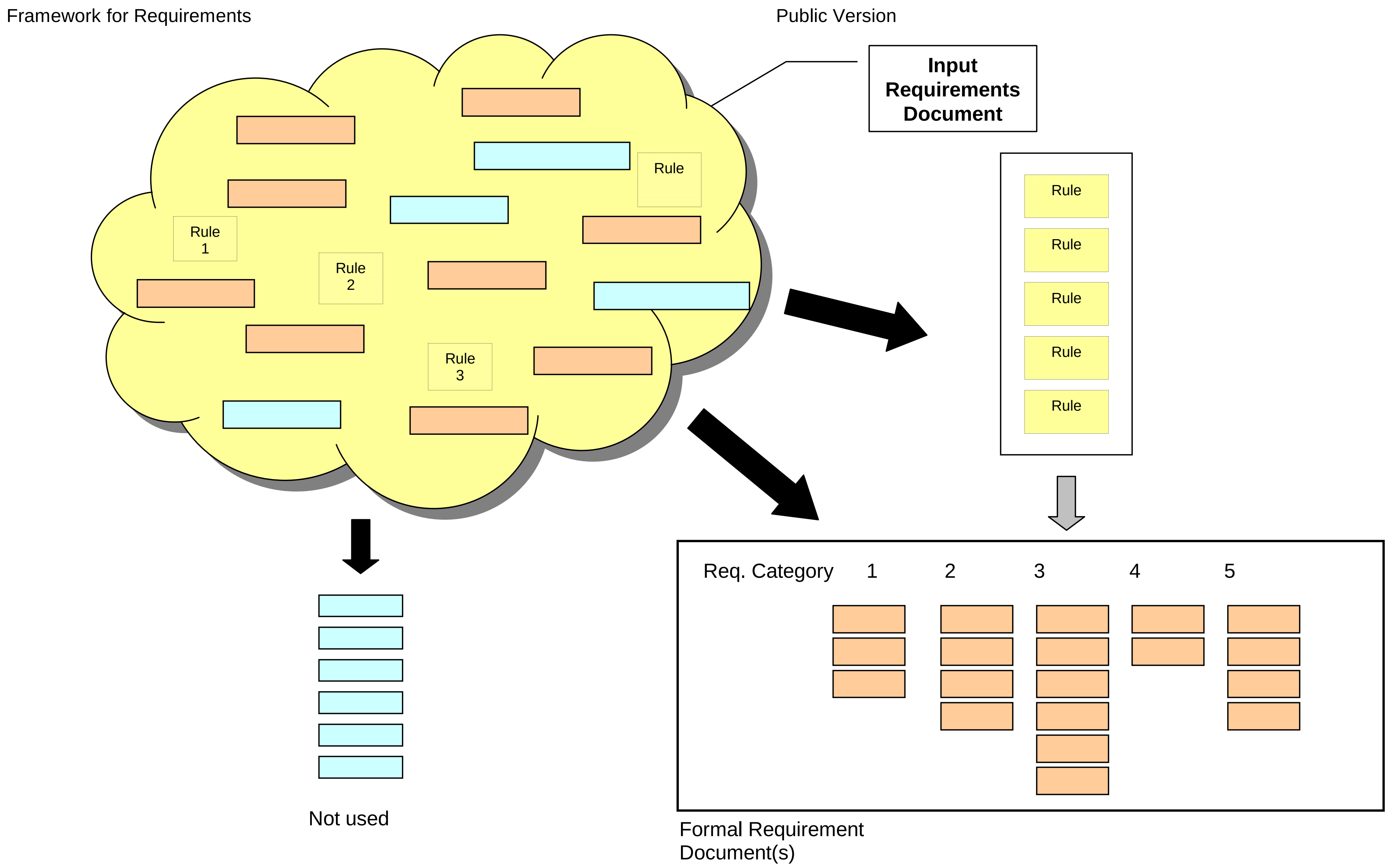
The aim of the present process proposal is to detail the work to be done in the area of “requirements identification” and “context/explanation identification”, which are represented by the shaded tasks identified in Figure 5-3. This figure shows another branch of activities, which is necessary in order not to loose any information in the discussed process, namely the link information between different requirements. However, this part of the process is dedicated to a separate study performed by HOOD and documented in [D3.2.2] chapter 5.1.

Early in most projects the predominant form that requirements take is text. This text could be the output of the elicitation techniques carried out before, and/or the text could consist of any technical documentation from a system to be modified, or from a system similar to that to be developed. A reason why text is so predominant is of course that it is the common language between the authority specifying the product and the organization creating it. The process described here will consider documentation that consists of text, but also tables, and figures. Further more a new development is regarded and not a project, which aims for changing an existing product.

It is not the intention of the process described herein to define the analysis techniques necessary to complete, refine and improve the original data covered by the input documentation. This is a specific field of activity not discussed here. The proposed process is solely concerned with reformatting and structuring input documentation into a form that allows the detailed analysis process to start.

Next page:

Figure 5-1: General goal of the study


Framework for Requirements
Public Version
Input Requirements Document
Rule
Rule 1
Rule 2
Rule 3
Rule
Rule
Rule
Rule
Rule
Not used
Req. Category 1 2 3 4 5
Formal Requirement Document(s)

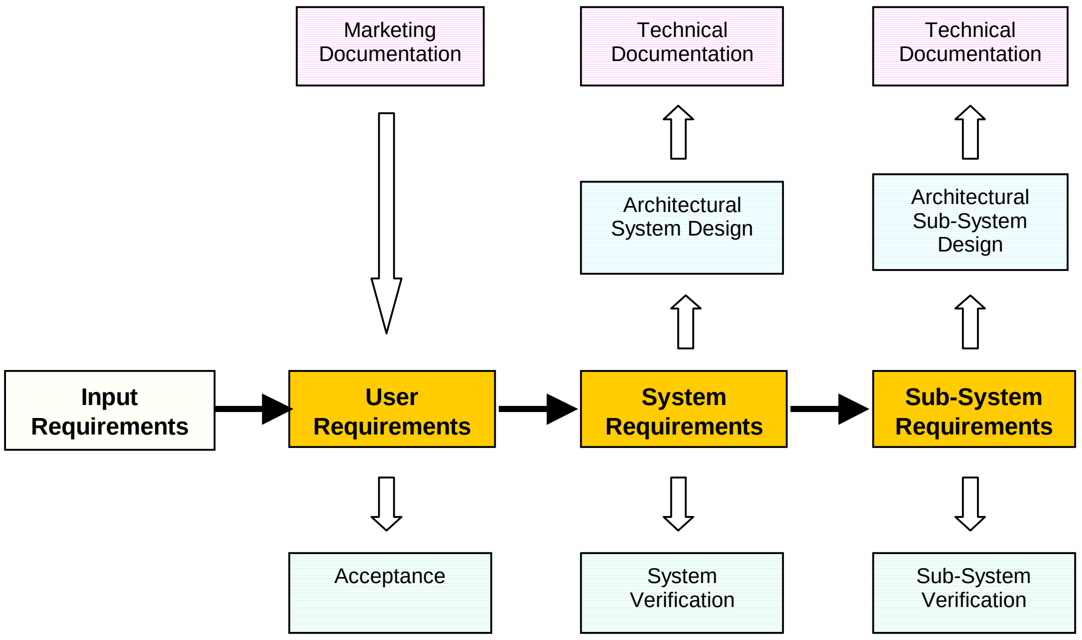


Figure 5-2: General Requirements Engineering Process

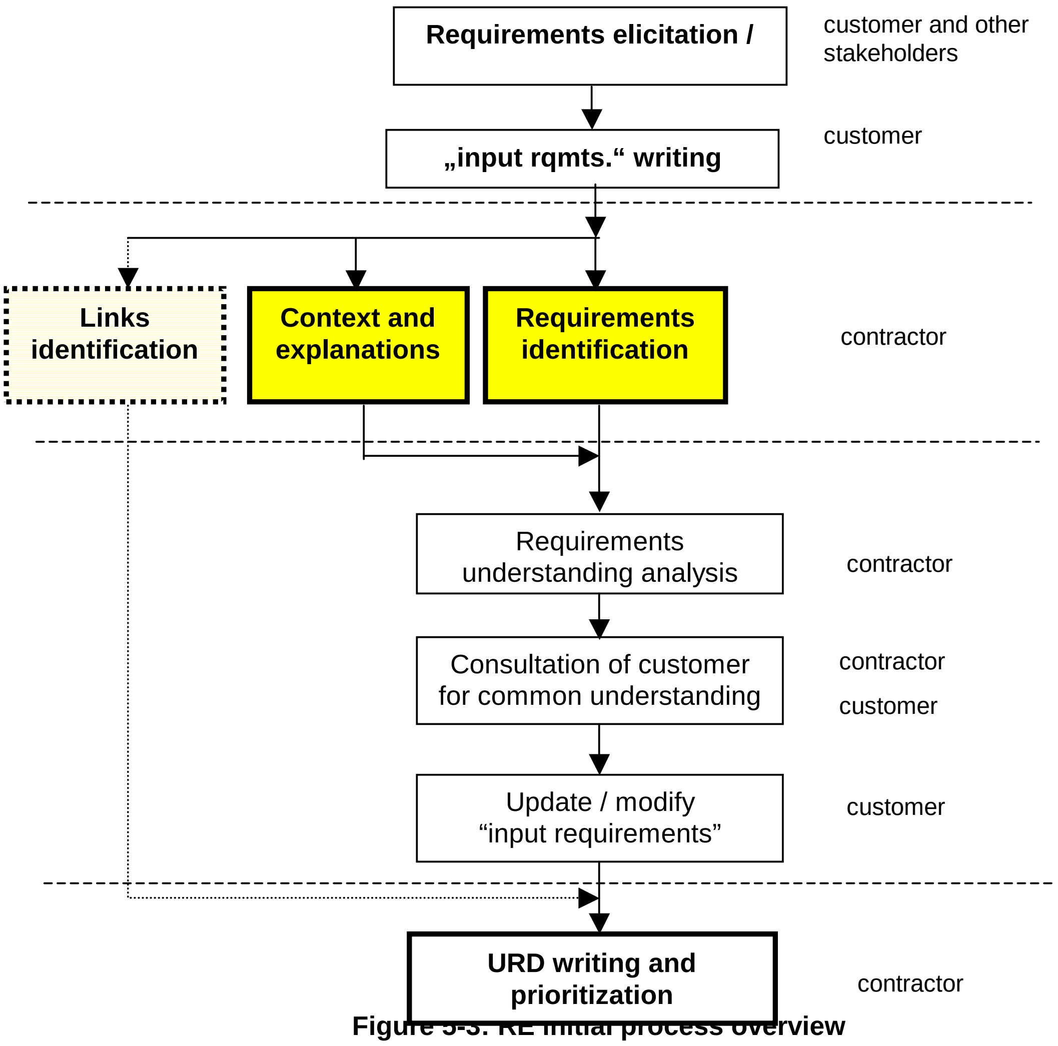

Requirements elicitation /
customer and other stakeholders
„input rqmts.“ writing
customer
Links identification
Context and explanations
Requirements identification
contractor
Requirements understanding analysis
contractor
Consultation of customer for common understanding
contractor
customer
Update / modify “input requirements”
customer
URD writing and prioritization
contractor

Figure 5-3: RE initial process overview

5.1.3.2 Customer and Stakeholders

Within the above section at several occasions both the customer and the stakeholders have been mentioned. They both have significant roles in the provision of the input requirements, and therefore some principal considerations are worth being discussed in this context.

Whenever an organization has got the opportunity and the task to develop a new system, or improve/change an existing system, there is normally somebody involved who wants the new system to be created and who intends to pay for it. This can be a party, which has at first instance nothing to do with the organization offering to perform the development. For example an aircraft company wants to introduce a new, electronically controlled system for the reduction of fuel consumption of the engines of their low cost 20 seat airplanes. In this case this company is the customer and has got a considerable set of wishes and requirements to put forward to the potential supplier, named the “contractor” in this study.

There are however quite a lot of other stakeholders who are keen to address their constraints and requirements. For example, the Ecological Parties have a strong interest to make sure, that the fuel consumption and the toxicity of exhaust gases is even better than required by the applicable laws. Other stakeholders are the airplane service companies maintaining and repairing the airplanes at airports. They want their task to be adequately supported. A third stakeholder is the company providing the engines, which for sure will be effected by any fuel consumption constraints. And there are other stakeholders as well, who are not all listed here.

This example shows, that there is potentially a set of input requirements documents valid for the contractor of the system. It is however the task of the customer to decide, which of them he is willing to accept, since he will be paying for it. The fact, that many requirements will most probably imply high development, production, and service cost, has the effect of less sales of the airplane, and therefore potentially less profit for the manufacturer. However, on the other hand this may serve the customer.

Thus, the customer will in this case have the lead of what will be the *input requirements* for the contractor/supplier, from different stakeholders. Due to this situation, the following sections are using the term “customer” only, which however implies, that other stakeholders are covered by this notion.

Figure 5-4 provides a schematic of the stakeholder-input-requirements-process described above. The figure points out, that there are stakeholders who are directly related to the project, as in the example case above the engine supplier and the maintenance organisations, and in addition the “affected stakeholders”, as the households who will be suffering from the exhaust gases and which are for example represented by an Ecological Party. It must be emphasized, that the customer himself is a stakeholder as well. He may be called the “Main Stakeholder”, since he makes the decisions and will be responsible for this decision in front of the other stakeholders.

After this decision a set of input requirements is available, which can be consolidated into one User Requirements Document. This is in principle the task of the customer, however is often “subcontracted” (but not always paid) by the customer to the contractor (for this reason the dotted line crosses the contract and the URD in the figure). The completed user requirements are then the technical basis for a development contract.

The above considerations do not cover the case, where a company decides by itself to develop a new product. In this situation there is some kind of a customer nevertheless: for example the marketing department or the company top management. With respect to input requirements and stakeholders this is principally the same situation as in the above example. The activities and responsibilities are the same; however there might not be a formal contract between the both parties.

It has to be pointed out, that later on, when the development contract is in house, there are further categories of stakeholders: company stakeholders and subcontractor/supplier stakeholders. These are however not relevant for the present input requirements process, since they do not contribute to the input requirements discussed in this study. They may however influence the system requirements to be established by the contractor, or at lower levels of contracts.

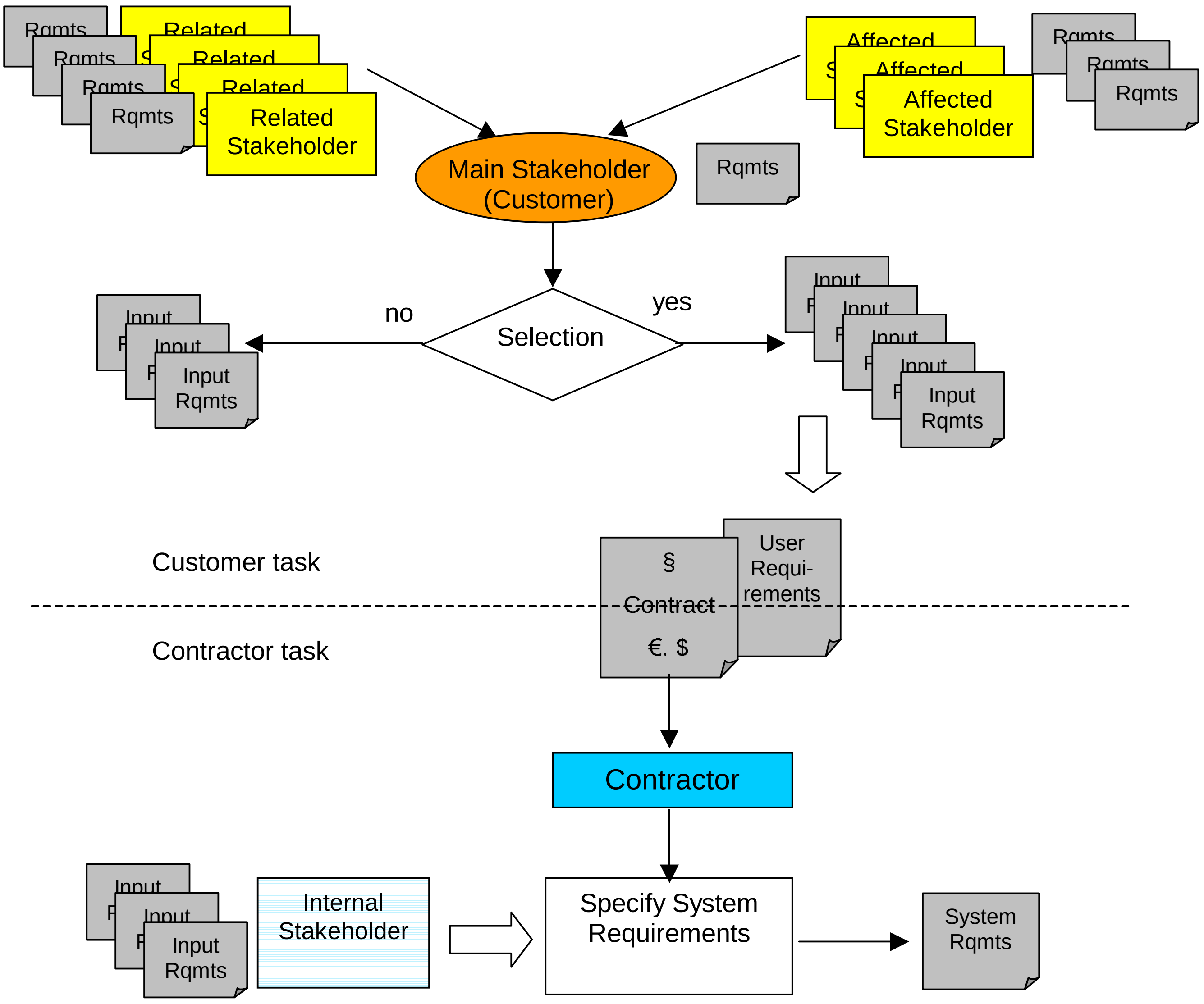


Figure 5-4 Stakeholders to customer relation

5.1.3.3 Input Documentation

It is understood that parts or all of the artefacts (documents and data) as listed below are provided by the customer and can be included under the definition of “Input Documentation”.

❑ Requirements	- all kinds of requirements issued by the customer
❑ Glossary	- definition of nouns, abbreviations, key words etc.
❑ Descriptive documents	- rough description of the future or a similar system
❑ Constraints and Scope	- data identifying the limitations of the future system
❑ Current system Documents	- all documents available from similar systems

❑ Correspondence Documents	- all documents like data faxes, letters, e-mails from stakeholders, containing requirements or wishes

It is considered that, as evident in a multitude of real cases, the requirement information is spread over a set of different documents, not all having fixed and well defined interrelation-ships. In addition it is assumed that the requirements are not put in a well-structured format and order, which is unfortunately quite often the case.

An example document for an input requirements document is provided at Annex A, namely the specification of a Motor Vehicle Wiper System. An automotive producer, who subcontracts his wiper system to a supplier, and tells him what functions the system shall have, issues it. We can see from this document that a typical source document is rich in information about the project, but does not identify clearly, which information relates to requirements and is identified as such by adequate words (must, shall, should etc.). However the information in the source document is not in a format that can be used by the requirements engineer. Before he can use this document he must first extract the requirements from it. He must then analyse them, understand their priority and context, and know where they came from. Then he must allocate requirements to the correct abstraction layer and further refine them.

Glossaries may be of interest, although they do not contribute to the definition of functions, performances and properties. This is due to the fact, that they contain definitions of the terms used in the input requirements documents, and by that can contribute to the correct understanding of the requirements. In addition they may help to categorize the functions of the system to be developed.

5.1.3.4 Output Documentation and Process Overview

Because the input documentation is written by the customer and therefore should not be altered by the contractor without negotiation, the concept presented will not change the input documentation. The goal is to analyse the input information structure in using verbal, grammatical, and text formatting attributes and then in a second step extract the categorized requirements data.

The process principal flowchart is provided in Figure 5-5: The input documentation enters the development process and is introduced in a partially populated requirements engineering model. In order to save the customer wishes and requirements this model is put under configuration control. From the customer provided, non-categorized data, the overall requirements engineering model for the system to develop is created. This is performed by completing this data by new sets of data, which might as well provide other “views” than presented by the input documentation. These views can be extracted into separate documents or data folders. The initial information however will not be changed in any way.

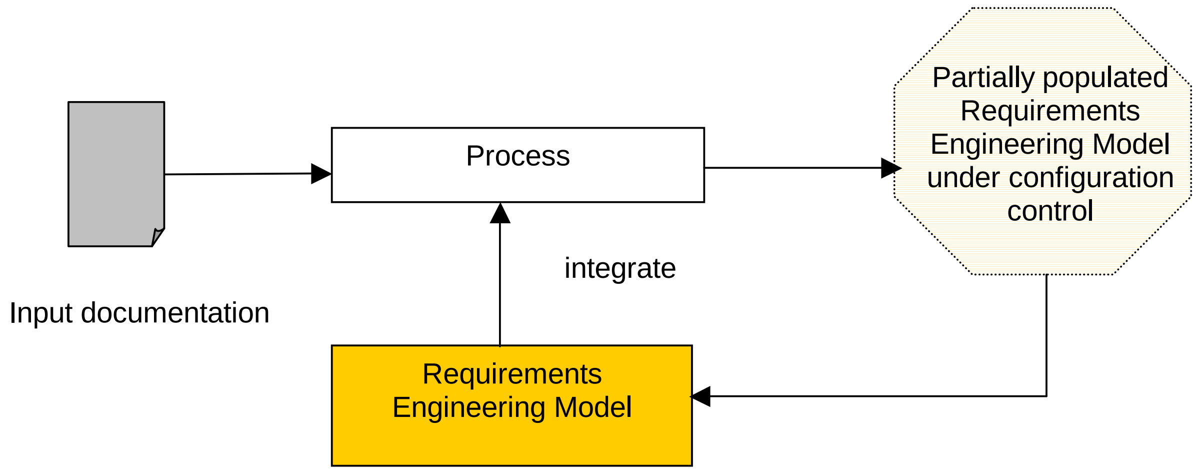

Process
Partially populated Requirements Engineering Model under configuration control
integrate
Input documentation
Requirements Engineering Model

Figure 5-5: Process principal workflow

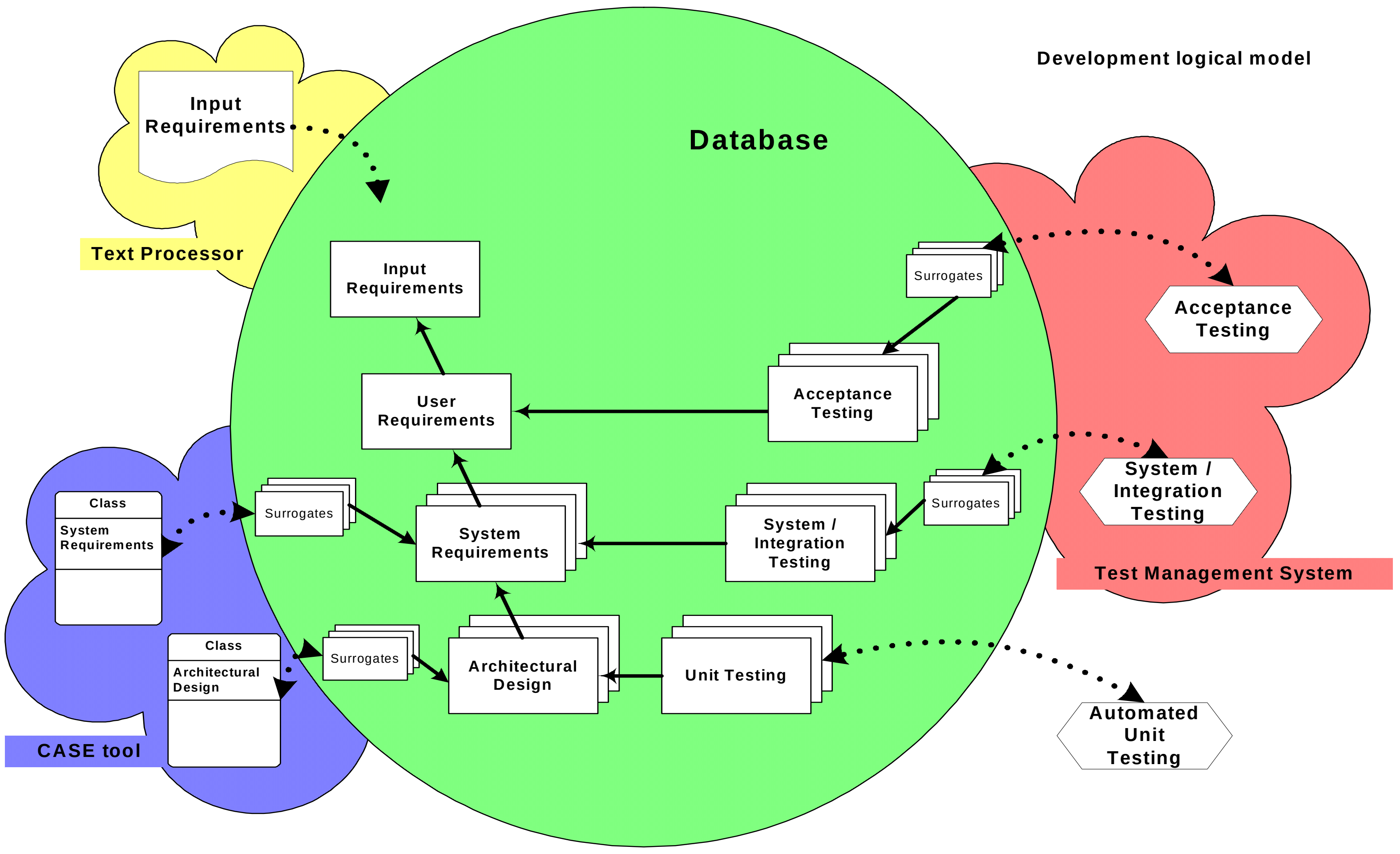
In order to know where to allocate requirements, the requirements engineer must first choose the requirements engineering model into which these requirements will be introduced. The requirements engineering model that he is going to use may well be defined by the systems engineering process of his organization. The requirements engineering model used in this study is generic. Whatever the difference between the model defined by the organization and the generic one described herein, the process is essentially the same.

The model described is a simplified version of the V-model, which is shown in Figure 5-6.

The process assumes that the import documentation is available in electronic form. It also assumes that the requirements engineering model will be stored in a database or in a requirements management tool. Such a tool will provide the necessary support for cross-references between the requirements and the change control utilities for later processing of the requirements. Note that this implementation requires the use of interfaces between the RM tool and other tools used during development.

Next page:

Figure 5-6: Development V-Model


Development logical model
Database
Input Requirements
Text Processor
Input Requirements
User Requirements
Acceptance Testing
Surrogates
Acceptance Testing
System / Integration Testing
Test Management System
Class
System Requirements
Surrogates
System Requirements
System / Integration Testing
Surrogates
Class
Architectural Design
Surrogates
Architectural Design
Unit Testing
Automated Unit Testing
CASE tool

5.1.4 Analysis of Available Source Documents

5.1.4.1 Starting point

A customer delivered an input documentation, gathering the requirements data from all accepted stakeholders, which may be not formulated sufficiently well for the purpose of an efficient requirements engineering: requirements are not easily and clearly identifiable, occasionally requirements are consisting of implementation solutions, which limit the contractor's ability to implement an optimised solution. In addition user, system and subsystem requirements may be mixed.

As already pointed out, the fact that the delivered requirements may not fulfil the quality criteria like completeness, consistency etc. is not the topic of the following works.

The main goal is to structure all those input information, which are input requirements, into a categorized form as described in Figure 5-1.

For the purpose of the present report only one official EMPRESS document, released for common use, was available as the database and was analysed in order to find out rules for categorizing. This report is titled as follows:

- Motor Vehicle Wiper System (no Documentation Number; see Annex A)

This means that only one author is available, which implies that differences in writing styles, vocabulary, and formatting styles are not covering the whole potential of individual habits thinkable.

It is pointed out that the analysis of the text has been performed under the sole aspect of requirements influencing software development. Where the document does not specify whether a software or hardware solution is aimed for, related assumptions have been made. In addition it has to be mentioned that by the fact of an English example document only principal rules could be elaborated, and a comparison between the behaviours and performances of the rules at different languages could not be performed. However, a general consideration on these topics is covered in the last sections of subchapter 5.1.

5.1.4.2 Goals of the Analysis

The goals of the analysis process performed are:

- ❑ Identifying comments and explanations which are definitely no requirements (C)
- ❑ Identifying User Requirements (UR)
- ❑ Identifying System Requirements (SR)
- ❑ Identifying Subsystem and lower level Equipment Requirements (SSR)
- ❑ Identifying Interface Requirements (IR)
- ❑ Identifying Design Requirements (DR)
- ❑ Identifying other requirements (oR)

The above categorization is not based on a specific technical domain or development process. This is due to the fact that it is not really relevant for the proposed method, and by the way should not be. The related definitions of these categories are provided in section 5.1.5.2.3.

As already mentioned earlier, it is possible to either filter out the non-requirements first, or the requirements first. This may depend on the tool used for processing the texts, and on the percentage of non-requirements among the requirements. This decision is however not of high importance.

5.1.4.3 Analysis Example and Related Assessments

5.1.4.3.1 Overview

For the purpose of the EMPRESS studies a motor vehicle related specification documents is available and can be used for analyses without any property or confidentiality restrictions. The document is available as a WORD file and describes the requirements for a car wiper system in English language.

The document allows analysing whether rules of categorizing the requirements can be applied for the English language. It must be pointed out that one document represents a singular writing style and thus cannot be representative for a multitude of cases. However, first analysis results may yield achievements for more detailed studies into the problem.

It is pointed out that tables within the above document have been analysed as well, although it is not possible to perform a text analysis containing grammatical features with this kind of information, since it consists of mere words and not of sentences. The same is valid for figures of any kind. It is however quite clear that at minimum tables generally contain important information, thus most probably requirements information. For this reason a table must always be analysed with respect to potential requirements.

5.1.4.3.2 Vehicle Wiper System

For the purpose of the EMPRESS studies this document has been translated from German to English. It is of a non-specific technical content, which is neither related to any specific vehicle brand nor to a specific series car.

For the purpose of the present study this document has been exported into EXCEL in order to ease work in a spreadsheet environment. This is beneficial for introducing attributes in columns.

The first paragraph of the document (chapter 1) is an introduction, which has no requirements management and engineering related contents, and is only an explanation of the information basis, and that it is especially produced to EMPRESS. Therefore the sentences of that introduction have been neglected by the following analysis.

The wiper system specified is not known in detail; especially it is not clear whether it is controlled via a data bus system. The document does not mention related interfaces. Since the EMPRESS study is concerned with embedded real time software systems, certain assumptions have been made in order to make this requirements document usable. The assumptions are as follows:

- All switch position mentioned are recognized by a bus system
- All sensor signals are recognized by a bus system
- All such data are processed via a dedicated processor
- The command outputs are created by this processor and control the power relays, which then operate the diverse electrical motors or actuators.

The above assumptions imply that a software package must be realized in order to perform the overall wiper system tasks. It is therefore assumed that the wiper specification also covers software aspects indirectly.

As a general remark for the document it must be stated that it does contain no information on design of any level (architecture, manufacturer, technologies etc.). Thus, rules for the related requirements could not be elaborated. The same is true for “other requirements” (oR). Related proposals are however made and must be verified with other kinds of input documents.

5.1.4.3.3 Analysis Approach

The steps of the analysis were fixed as follows:

1. Export the text (see Annex A) into EXCEL (Remark: it is possible to perform the work in DOORS, however there were administrative reasons to use EXCEL). The columns created are visible from Table 5-5.
2. Identify which text parts are requirements, or non-requirements Comments (C) (column 3). Identify related rules.
3. Identify which of these requirements are related to software (column 4). This step is specific for the available working document (Wiper System, Annex A)
4. Identify the abstraction layer that seems appropriate from a technical point of view: Categorize the software requirements into User Requirements (UR), System Requirements (SR), Subsystem Requirements (SSR), Interface Requirements (IR), Design Requirements (DR), and other Requirements (oR) (column 5).
5. Search for rules that allow identifying the different requirements categories. Verify explicitly for each single information object whether it can be categorized via the defined rules, and whether misinterpretations take place by applying the defined rules.

In order to provide some view into the analysis method described here, a sample of the EXCEL table as quoted in step 1 above is shown in Table 5-3 here below.

Next 3 pages:

Table 5-3 **Sample of the Reference Input Document, imported into EXCEL for analysis**
(only those areas, which are quoted in the text)

ID	Text Element	Rqmt.	S/W related	Abstraction Layer	Key words UR	Key words SR	Key words SSR	Comments
InR 34	The rear wiper motor only has one speed level.	x	x	SR		motor		
InR 35	It therefore only has one relay:	x	x	SSR			one	
InR 36	Rear Wiper Relay ON/OFF	x	x	SSR			ON, OFF	
InR 37	**Rear Wash Pump (only for S)**							
InR 38	(See front wash pump above)	x	x	SSR		pump		misinterpretation due to reference
InR 39	**Other sensors involved**							
InR 40	**Speed sensor**						speed sensor	
InR 41	Signals the current speed.	x	x	SSR		speed		misinterpretation due to splitting of Header and requirement
InR 42	**Gear recognition**							
InR 43	Signals the gear being engaged.	x	x	SSR		gear		misinterpretation due to missing of "sensor"
InR 44	**Functionality**							
InR 45	**Front Wipe and Wash Function**							
InR 46	Precondition:							
InR 47	Clip 15R „ON“	x	x	SSR			clip, ON, 15	
InR 48	The functions are broken off at clip 15R „OFF“.	x	x	SSR			clip, OFF, 15	
InR 49	**Tip Wipe Function**							
InR 50	The tip wipe function has priority over the intermittent wipe function.	x	x	UR	wipe, function			
InR 51	If the intermittent wipe function is simultaneously activated during tip wipe function, the interval time is restarted after the tip wipe function has finished.	x	x	UR	wipe, function			
InR 52	The tip wipe function is recognized during "level 1" or "level 2"	x	x	SSR			1, 2	
InR 53	Briefly pressing the steering column switch activates a single wipe action	x	x	SR		switch		
InR 54	If the switch is kept pressed in this position the wipe function will continue as long as it is held down.	x	x	SR		switch		
InR 55	The tip wipe function takes place independently of the wash pump operation.	x	x	SR		pump		
InR 56	**Wash Function**							
InR 57	The wash function is actuated if the steering column switch is pressed to the position after the tip wipe function position.	x	x	SR		switch		
InR 58	The wash function continues as long as the switch is pressed down.	x	x	SR		switch		
InR 59	If the wash function is active during “Level 2”, wiping takes place in “Level 2”.	x	x	SSR			1, 2	
InR 60	**Rewipe Function**							
InR 61	The rewipe function becomes active once the pump stops working and the wipe cycle is complete.	x	x	SR		pump		
InR 62	If the actuation of the wash pump takes place for t<1s, the rewipe action is activated once.	x	x	SR		pump		
InR 63	Activating t>1s repeats the rewipe action three times.	x	x	SSR			time, s	
InR 64	An additional activation of the wash function during the rewipe function will be ignored	x	x	UR	wash, rewipe, function			
InR 65	If the rewipe function is active during “Level 2”, rewiping takes place in this level.	x	x	UR	rewipe, function			

ID	Text Element	Rqmt.	S/W related	Abstraction Layer	Key words UR	Key words SR	Key words SSR	Comments
InR 66	Rewiping during "Level 1", "Intermittent", or when the steering column switch is in "Position 0" (=OFF) takes place in "Level 1".	x	x	SSR			OFF, 1	
InR 67	The rewipe function is executed independently of the subsequent requirements of the steering column switch	x	x	SSR			steering column switch	
InR 68	**Intermittent Wipe Function**							
InR 69	If the intermittent wipe function is transmitted from "Position 0" (=OFF) the wipe process will take place immediately.	x	x	SSR			OFF, 0	
InR 70	After the intermittent wipe function has been interrupted by a wipe function, the interval delay will rerun.	x	x	UR	wipe, function, rerun			
InR 71	The intermittent wipe function always operates in "Level 1".	x	x	SSR			1	
InR 72	At a speed below 50 km/h the interval time is t=5s+/-1s.	x	x	SSR			50, km/h, 5, 1, s	
InR 73	Above 50 km/h it is t=3,5s+/-1s	x	x	SSR			50, km/h, 3, 5, 1, s	
InR 74	**Wipe Level 1**							
InR 75	The wipe relay "ON/OFF" is actuated, i.e. continuous wiping at wipe speed 1.	x	x	SSR			ON, OFF, 1	
InR 76	The wipe process will not be influenced by additional tip wipe or rewipe functions.	x	x	UR	wipe, rewipe, process			
InR 77	**Wipe Level 2**							
InR 78	Both relays (relay "ON/OFF and relay "Level 1 / Level 2") are actuated, i.e. continuous wiping at wipe speed 2.	x	x	SSR			ON, OFF, 1, 2	
InR 79	The wipe process will not be influenced by additional tip wipe or rewipe functions.	x	x	UR	wipe, rewipe, process			
InR 80	**Wipe Function Completion**							
InR 81	If the wiper arm is outside the basic position when the wipe function is switched off, the wipe function will finish once the arm is back in its normal position, i.e. until clip31B-V has changed from "HIGH" to "LOW":	x	x	SSR			clip, high, low, 31	
InR 82	For clip 15R "OFF", the wiper remains in its current position.	x	x	SSR			OFF, clip, 15	
InR 83	**Rear Wipe Function and Wash Function**							
InR 84	Precondition:							
InR 85	Clip 15R "ON"	x	x	SSR			clip, ON, 15	
InR 86	The functions are broken off at clip 15R "OFF".	x	x	SSR			clip, OFF, 15	
InR 87	**Intermittent Wipe Function**							
InR 88	The rear wipe function is activated by the rear wiper switch and takes place in the intermittent mode.	x	x	SR		switch		
InR 89	An additional activation of the rear wiper switch turns the function off.	x	x	SR		switch		
InR 90	Actuation time: t=0,5 s (+0,3 s/-0,1 s)	x	x	SSR			0, 1, 3, 5, s	
InR 91	Interval time: t=5 s +/- 1 s	x	x	SSR			1, 5, s	
InR 92	**Wash Process**							
InR 93	Keeping the rear wiper switch pressed down activates the wash process.	x	x	SR		switch		
InR 94	The wash process has priority over the intermittent wash process and continues until the rear wiper switch is released.	x	x	SR		switch		
InR 95	An already activated intermittent wash process is not stopped by the wash process	x	x	UR	wash, process			

ID	Text Element	Rqmt.	S/W related	Abstraction Layer	Key words UR	Key words SR	Key words SSR	Comments
InR 96	**Rewipe Function**							
InR 97	The rewipe function is activated as soon as the pump stops working and the wash cycle is complete.	x	x	SR		pump		
InR 98	The rewipe action is then repeated twice.	x	x	SSR			twice	
InR 99	**Wipe Function in Reverse Gear**							
InR 100	The rear window wiper is automatically activated in reverse gear if the front wipers are on.	x	x	SR		gear		
InR 101	If the windshield wipe function is in “Level 1” or “Level 2” the rear wiper will go into the continuous wipe mode.	x	x	UR	wipe			
InR 102	If the windshield wipers are in the intermittent mode, the rear wiper will wipe intermittently	x	x	UR	wipe			
InR 103	The wipe function in reverse gear has priority over the intermittent wipe function.	x	x	SR		gear		
InR 104	The rear wipe function will be deactivated if the reverse gear is disengaged or the front wiper system deactivated.	x	x	SR		gear		
InR 105	**Wipe Function Completion**							
InR 106	If the wiper arm is not in its basic position when the wipe function is switched off, the arm will automatically complete the wipe function and return to its basic position.	x	x	SR		wiper, arm		
InR 107	The wiper arm will be in the basic position if the wiper cam has been reached, i.e. clip 31B-H changes from "HIGH" to "LOW"	x	x	SSR			clip, 31, high, low	
InR 108	At clip 15R “OFF” the wiper will remain in its current position.	x	x	SSR			clip, 15, OFF	

5.1.4.3.4 Step 1

Task: Export the text (see Annex A) into EXCEL. The columns created are visible from Table 5-3.

For this activity there are no specific rules to follow except what is necessary for the proper operation of these programs.

5.1.4.3.5 Step 2

Task: Identify which text parts are requirements, or non-requirements Comments (C) (column 3). Identify related rules.

Within the second activity it has been identified whether a sentence or piece of text is a requirement or a header, comment or explanation. The following rules have been identified and applied (sufficiently consistent rules are written in bold within a shaded text box, vague rules are written in normal fonts, within a smaller text box):

Rule 2.1 **A piece of text can be a header, if no verb is used. Requirements text contains verbs (generally)**

Assessment: this rule may not be successful in all cases, since the input document authors used rudimentary sentences or pieces of text also for requirements (see for example chapter 1.1.1 “Wiper Relay On/Off” etc.)

Rule 2.2 Headers may be written in bold fonts

Assessment: this rule may not be successful in all cases, since the input document authors used bold fonts also for requirements (see for example chapter 1.1.1 “Wiper Relay On/Off” etc.)

Rule 2.3 Headers may be written in larger fonts

Rule 2.4 Headers may have increased spacing to the previous and / or subsequent text

Rule 2.5 Headers may have a numbering or an enumeration sign (dot, dash, bullet, square, arrow etc.)

Rule 2.6 **All texts between Headers can be either Requirements or Comments / Explanations**

Rule 2.7	**Each single piece of text, be it separated to the adjacent ones by full stop, TAB, line feed, or page feed, are objects which can either be a requirement or another text.**

Rule 2.8	**Tables contain multiple objects and are considered to be requirements**

Rule 2.9	**Figures may contain requirement information or explanatory information; this is to be decided by quotations in the text and/or by legends or callouts**

Assessment: figures may well contain requirements information, especially if they contain mathematical, physical, or detailed technical data of any kind.

This step represents in principle the main part of the function, which the import function of DOORS performs whenever a WORD document is transformed into a DOORS Formal Module. This fact can be used for the definition of a Demonstrator. It is beneficial to make use of DOORS in this case in order to abbreviate the work necessary for the classification of requirements / non-requirements. It has however to be mentioned that the function, which DOORS provides has to be checked by the requirements analyst or a system engineer.

Although DOORS provides this function, the aim of the first analysis is to see which tasks the analyst has to perform, how difficult they are, and how long they take. The method should make sure, that the initial problem of this study can also be resolved when not having DOORS available.

5.1.4.3.6 Step 3

Task: Identify which of the requirements are related to software (column 4). This step is specific for the available working document (Wiper System, Annex A)

For this step no specific rule was necessary. Based on the assumption made in paragraph 5.1.3 all but one requirement (InR 12, see Table 5-3) is relevant to the software development.

5.1.4.3.7 Step 4

Task: Identify the abstraction layers that seem appropriate from a technical point of view.

In the present example it was appropriate to categorize the software requirements into User Requirements (UR), System Requirements (SR), Subsystem Requirements (SSR), Interface Requirements (IR), Design Requirements (DR), and other Requirements (oR).

These categories have been chosen on the basis of the classification baseline of chapter 2.3, and on the basic knowledge in the domain of automotive systems. The rules used hereby cannot easily be noted down since they rely on the technical education and professional experience of the analyst. However, the essential of this step is to assess the level of detail used in the information analysed. Low technical detail tends to be a User Requirement, high technical detail tends to be a Subsystem Requirement.

5.1.4.3.8 Step 5

Task: Search for rules that allow identifying the different requirements categories. Verify explicitly for each single information object whether it can be categorized via the defined rules, and whether misinterpretations take place by applying the defined rules.

Searching for grammatical and verbal correlations between the abstraction layers and the information itself performed this task. Due to the fact that many of the available information elements were not presented in form of sentences, the search for grammatical attributes was not successful. Therefore the analysis of words used in the text has been intensified. At first nouns and verbs have been considered, since they contain the basic information in a sentence. Due to the fact that this first approach has already been successful, adjectives, adverbs, and other words have no more been considered. This is however not considered as a prove, that these omitted words cannot contribute to the categorization of the information. Further studies may find out which benefits may be gained from these elements of language information.

As generally known, adjectives and adverbs can be an indication for non-functional requirements, or for a link to another bit of information. This was however not the case in the reference document.

The rules found for categorizing the requirements are:

Rule 5.1	**A requirement is a UR if the text contains only nouns and verbs not belonging to a specific technical domain. They only are related to general physical parameters, to general actions and stati.**

Glossary	see Table 5-3, column 1
Misinterpretations encountered:	None

Glossary	see Table 5-3, column 2
Misinterpretations encountered:	None

Rule 5.3	**A requirement is a SSR if the text contains technical, numerical, and physical details, domain related specialist terms, detailed stati and modes**

Glossary	see Table 5-3, column 3

Misinterpretations encountered:

- InR 30 does not contain appropriate words for rule 5.3, since the notion “tip switch” from the header has not been taken over by the author to the requirement text itself
- InR 38 has been misinterpreted due to the fact that there is only a text reference

Assessment: this rule is embracing a wide set of nouns and verbs which are specific to a technical domain, as well as figures and numerals, logical states, operational modes etc. This effort is considered requiring a medium work effort and advanced domain knowledge.

Rule 5.4	**A requirement is an IR if nouns appear which do not belong to the system’s product tree or functional content, however to external systems**

Glossary see Table 5-3, column 4

Misinterpretations encountered:

The two IR cases encountered in the reference document concern the car speed sensor, and the gearbox position sensor. Both of these are not part of the wiper system, however of the motor and the gear. In some cases (InR 34, 72, 75, 78, 100, 103, and 104) the author has used the key words “speed” and “gear”, without speaking about interface issues. These cases have been nevertheless correctly categorized due to rule 5.8 below.

Rules for DR could not be defined due to a lack of related information in the document. However, it is thinkable that the following rule be used:

Rule 5.5	A requirement is a DR if dedicated names of companies and product numbers or denominators, material qualities, design standards etc. are used.

Glossary Polyurethan, Siemens, DIN, ISO9001, VDE etc.

Assessment: this rule is embracing a wide set of nouns, which are specific to a technical domain, as well as names, alphanumeric words etc.; they can in principle be gathered in a list by the analyst of a certain domain, against which the text would be parsed. This effort is considered high.

Rules for “other Requirements” (oR) could not be defined due to lack of related information in the document. However, it is thinkable that the following rule be used:

Rule 5.6	A requirement is an oR if the information has been identified to be a requirement (rules 2.1-2.9), however does not contain keywords of categories UR, SR, SSR, and IR.

Assessment: This kind of requirements will have to be analysed carefully, since it was not able to be associated to one of the defined categories. It may well be, that the requirement is wrongly worded, not consistent, or not correct.

The following difficulty of the above procedure has to be resolved by an appropriate method:

Due to the fact that nouns and verbs of the UR and SR texts can be enclosed as well in SSR texts, the requirements analyst or an automatic parser will find for certain sentences / text pieces multiple solutions. This is shown via the following example (Table 5-4), where the above glossaries have been used:

Text	UR	SR	SSR	IR	DR
The intermittent wipe function of the wiper motor always operates in “level 1”	x	x	x		
Relevant key word from glossary:	Intermit-tent wipe function	Wiper motor	Level 1		

Table 5-4 Example for a multiple mapping case related to rule 5.7

In this case the lowest level (SSR) has always priority. This is justified, since the lowest level information “Level 1” is dedicated to the characteristics of the wiper motor control unit. Thus, it is the specification for the motor control, and not for the motor, nor for the intermittent wipe function, which are all higher levels of abstraction. This leads to another rule to be followed:

Rule 5.7:	**In case the parsing result is ambiguous, the result of the lowest level takes precedence.** **Order of precedence (increasing priority): UR, SR, SSR, IR, DR**

A second difficulty concerns the interface requirements: The reference document uses words of the IR-Glossary at places, where it definitely does not intend to speak about interfaces. However, the related information elements (sentences or other) need to mention these words since the related data are necessary at this place. For example, in InR 100 the word “gear” is used, since the wiper system needs the information that the gear is in reverse position. This implies that the rear wiper is automatically switched ON, in case the front wipers are ON as well. Thus, the gearbox needs to provide the signal “gear on reverse” via the data bus system. Nevertheless, this information is no interface information.

From that situation the following rule has been defined which solves the problem:

Rule 5.8 **If interface related key words are used in the information under analysis, and in addition key words of the categories UR, SR, or SSR are present, then the information is considered not to be an Interface Requirement, but of the category as defined by rule 5.7.**

Example InR 100:

Text	UR	SR	SSR	IR	DR
The rear window wiper is automatically activated in reverse gear if the front wipers are ON.		x		(x)	
Relevant key word from glossary:		Wiper		Reverse gear	

Table 5-5 Example for a multiple mapping case related to rule 5.8

The following Table 5-6 represents the glossary used for the text analysis of the present section:

UR	SR	SSR	IR	DR	oR
action	actuation	0	(car) speed	no entries	no entries
activation	delay	1	gear		
function	interval	2			
position	motor	3			
priority	pump	5			
process	stop	15			
rewipe	switch	<			
time	wiper	=			
wash		>			
wipe		cam			
		clip			
		high			
		high			
		km			
		low			
		mode			
		OFF			
		ON			
		one			
		relay			
		single			
		speed			
		steering column switch			
		steering column switch			
		three			
		tip-switch			
		twice			
		two			

Table 5-6 Glossary of words used for the text analysis

5.1.4.4 Analysis Results

The analysis performed “by hand” and without any support by parsing computer programs yielded an interesting set of findings. These are rules, which are able to cover a high percentage of all information pieces, in being able to categorize them into abstraction levels, which are useful for the further processing of the input requirements from the customer. Even very dense information elements like information formatted in a table can be treated with the same rules.

The EXCEL table has undergone a parse check, in order to find out whether the initial categorization by technical knowledge (Table 5-3, column 5) has been performed correctly.

The results of this analysis are:

- There were no misinterpretations for the User Requirements
- There were no misinterpretations for the System Requirements
- There were two (2) misinterpretations for the Subsystem Requirements
- There was no misinterpretation for the Interface Requirements

- Design Requirements have not been contained in the Input Document
- Other Requirements have not been contained in the Input Document

In order to get a numerical assessment of the above results the percentage of misinterpretations has been calculated:

SSR: 2 misinterpretations out of 46 (= 4.3%)

This success rate is very high and may be questioned. The authors' opinion is that the following expansions of the present study have to be performed in order to gain some more credibility:

- Analyse a set of various input requirements documents, from different technical domains, written by different authors.
- Perform the initial categorizing (step 4) within a team of experienced systems engineers, in order to improve the consistency of the assessment reference.

5.1.4.5 Alternative Rules

The above approach has been found as result of a detailed analysis process. It has allowed finding rules, which seem to be applicable to a variety of different technical cases and documents. It is however to be mentioned that rules of different character have already been recognized by experienced RE practitioners within HOOD GmbH. These rules are based on general grammatical and formatting features and analyse very general words not specific for a domain.

These rules are generally used implicitly by everybody when reading input requirements documents. They may be valid and should be analysed whenever the above detailed method seems not to be successful.

During the work performed it was quite evident that these alternative rules were not as general and powerful and did not lead to significant successes when applying them for the Wiper System reference document (Annex A). For that reason they have not been used. They are nevertheless provided here for general information.

Related to Step 2: Identification of Requirements and other texts:

Rule X1:	A requirement contains the word "shall" or "will" or "must"
Rule X2:	There are alphanumerical indicators near to the requirement, for example "**REQ 02871** The system…….."
Rule X3:	Sentence grammatical structure (Use of passive in present tense etc.)
Rule X4:	Located in a table
Rule X5:	Special formats as bullets or numbering in the used text editor

Related to Step 4: Identifying Requirements category:

Rule Y1:	Using certain words like “user”, “system”, “subsystem”, “equipment”, “component” etc.
Rule Y2:	Using words like “interface”, “design”, or other words which directly indicate, that a certain category is addressed (in this case either IR or DR).

5.1.5 Proposed Process and Related Rules

5.1.5.1 Process Overview

The process deduced from the achievements of the text analyses is quite simple and straightforward. Due to the fact that very few generally applicable rules could be identified, it has become clear that it must become a long-term aim of requirements management and requirements engineering to make sure that a standardized language should be used for the different technical domains.

The aim of the performed study is to

- Prepare the consultations between the contractor and the customer, which will normally be necessary due to the fact that the delivered input documents are not ideal in certain respects (EMPRESS RE-process sub-activity “Common Understanding” and “Negotiation”)
- Prepare the task of the contractor to perform a detailed requirements assessment, which identifies omissions, misunderstandings, unclear issues, glossary of terms, etc. (EMPRESS RE-process sub-activity “Verification and Validation”)
- Identify whether it is possible to find rules, which are sufficiently robust to be used for different authors, languages, and technical domains.

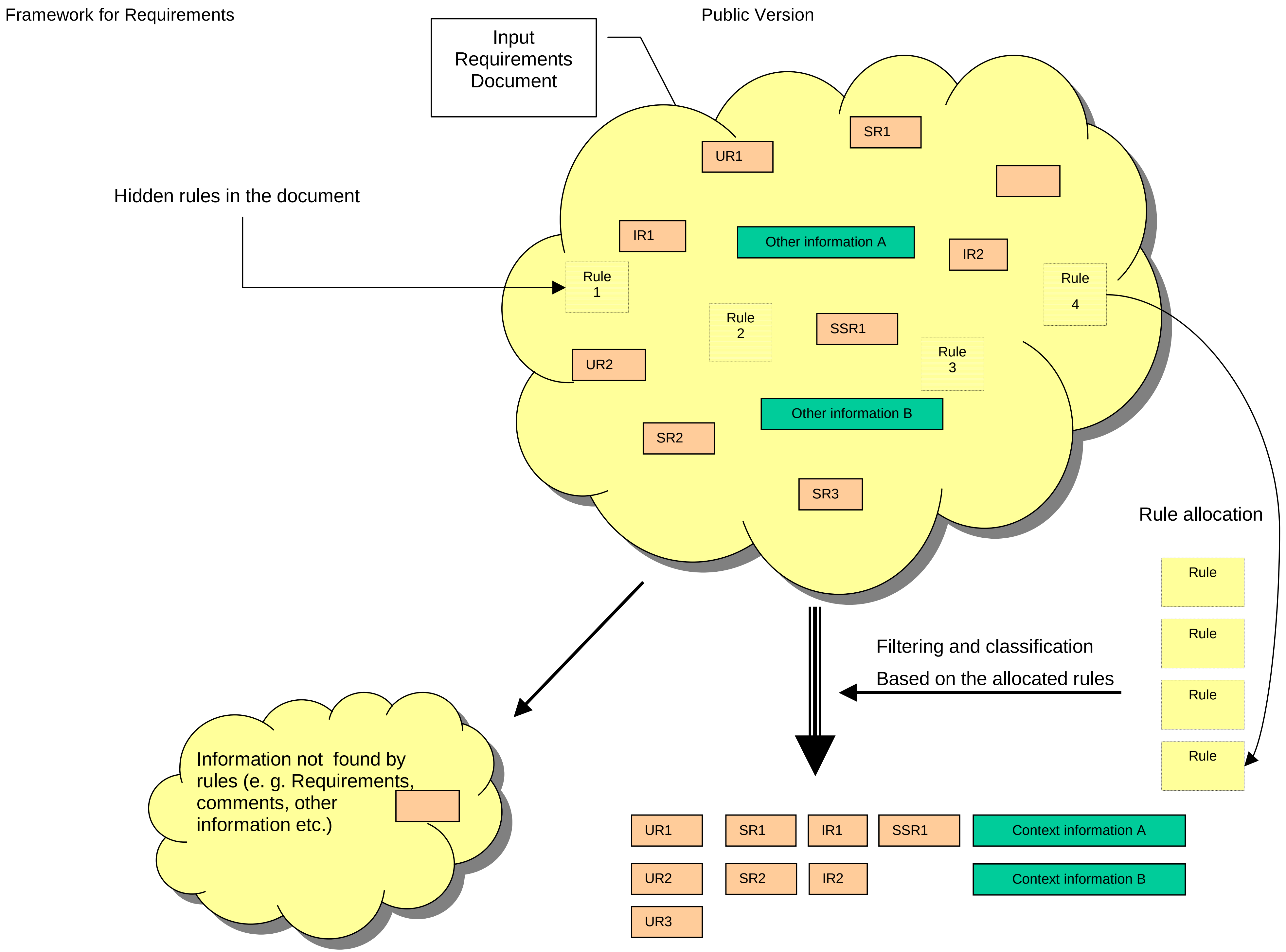
Figure 5-7 provides an overview of the process proposed, developed from Figure 5-1

Figure 5-8 shows the process in form of a simple flow diagram. A description of each activity in the process is provided in the following paragraphs. The process may look linear, but it needs not be followed in linear fashion; the activities described below could be carried out simultaneously where possible from a teamwork splitting point of view.

The process consists of a configuration control at the very beginning, followed by the attempt to format the document(s) such that they can best be further processed within a Requirements Engineering tool. The documents are now ready to be analysed sentence-by-sentence, header-by-header, and bullet-by-bullet. This is performed reading these elements and making decisions, whether the element under consideration is a requirement or not. This task performed, the identified requirements can be filtered out and classified based on the categories as listed in section 5.1.4.2.

Next page:

Figure 5-7: Process Overview



Framework for Requirements
Public Version
Input Requirements Document
Hidden rules in the document
UR1
SR1
IR1
Other information A
IR2
Rule 1
Rule 2
SSR1
Rule 3
Rule 4
UR2
Other information B
SR2
SR3
Rule allocation
Rule
Rule
Rule
Rule
Filtering and classification
Based on the allocated rules
Information not found by rules (e. g. Requirements, comments, other information etc.)
UR1
SR1
IR1
SSR1
Context information A
UR2
SR2
IR2
Context information B
UR3

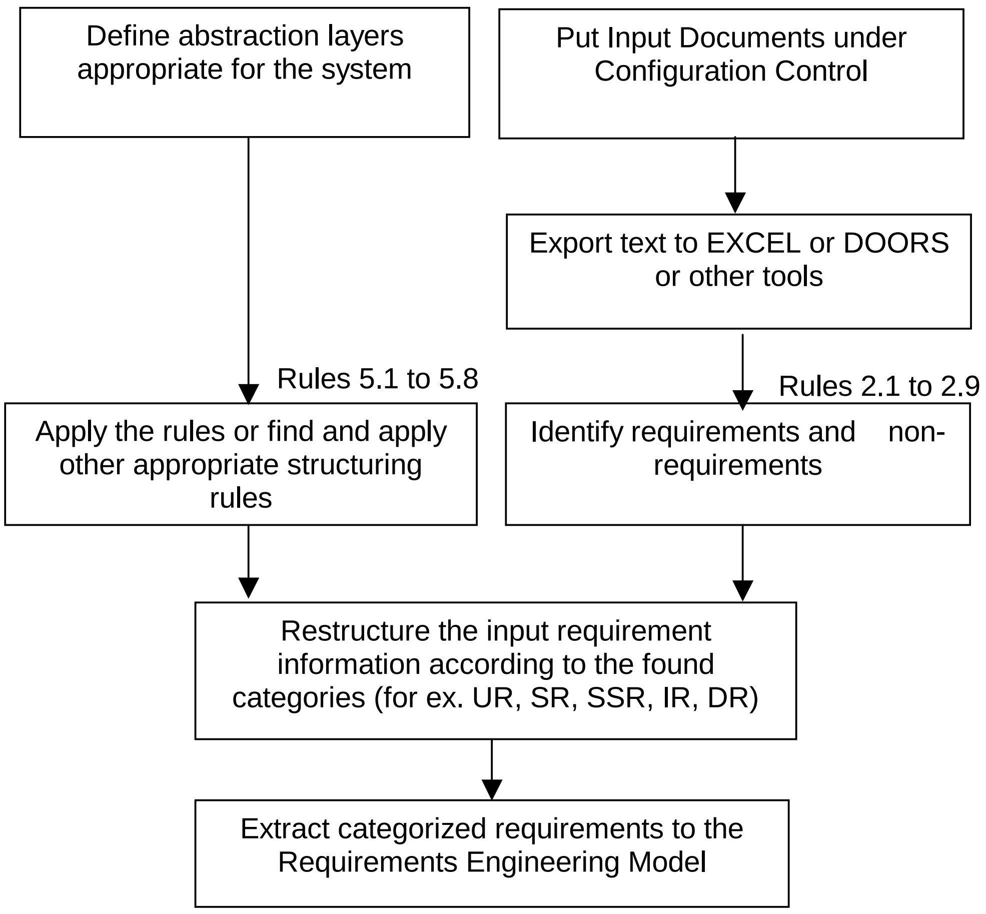

Define abstraction layers appropriate for the system
Put Input Documents under Configuration Control
Export text to EXCEL or DOORS or other tools
Rules 5.1 to 5.8
Rules 2.1 to 2.9
Apply the rules or find and apply other appropriate structuring rules
Identify requirements and non-requirements
Restructure the input requirement information according to the found categories (for ex. UR, SR, SSR, IR, DR)
Extract categorized requirements to the Requirements Engineering Model

Figure 5-8: Process flow for categorizing of initial customer requirements ("Input Requirements")

5.1.5.2 Process Steps

5.1.5.2.1 Input to the process

The present RM&E activity is part of the EMPRESS process, namely the Requirements Engineering discipline and stands at the very beginning of these efforts. It is part of the elicitation/collection and analysis/classification sub activities.

The first task must of course be to make sure that all relevant Input Requirements Documentation is available from the customer and all other authorized stakeholders. In case there is a contract already signed, this should be already assured. The inputs to this task are any kind of initial requirements information. Its form may vary a lot, however is not relevant. It can be communication notes as for example e-mails, data faxes, technical notes, minutes of meetings, protocols from interviews, descriptions and any other written information thinkable. If some of the input information should not be available in electronic form, it then should be scanned and OCR (optical character recognition) processed from printed information, or typed in from verbal information or handwritten notes. This is necessary in order to be able to apply functions available in office tools, thus supporting the overall activity described here.

5.1.5.2.2 Configuration control

The first activity after the electronic filing above must of course be to make sure that all relevant Input Requirements Documentation is base lined and protected from non-authorized and uncontrolled changes. The documentation received must therefore immediately be placed under some form of configuration and change control. This makes sure that all team members involved in the further work with these documents can be sure that they work with the right and recent set of document issues.

For each document it has to be made sure that it is uniquely named and that the latest version is available.

It is beneficial to ask the supplier of an input requirements document for a glossary. This is necessary for the understanding of the input documents and forces him to think very carefully about what he is writing and requiring.

5.1.5.2.3 Formatting the input documentation

Quickly read each document trying to get an overview of its context and content. Try to estimate how easily they can be manipulated into the form required by your requirements engineering database. It may be that the authors of the documents require to use their standard format, in which case, after importing into the requirements engineering model, there will have traceability to the original document to be maintained.

An important decision at this point is the granularity with which each document will be analysed. Will it be necessary to reference each individual sentence in the document or is it sufficient to reference the whole document itself? It is common to reference whole documents when they provide background information, but the documentation that provides requirements is usually broken into sentences.

The rest of this process assumes that all input documentation will be broken down into individual sentences. Each of these individual sentences will need to be uniquely identified. However it must be considered that there is a structure implied by the internal document hierarchy. If the documents are simply broken into individual sentences one could lose the context and the structure that is implied by the headings in the document.

Different requirements management tools and databases require different input formats. Some requirements management tools allow importing directly from WORD or other text documents, some tools provide an interface between WORD and a database, but most databases expect the data in *.csv format. For this reason one should consider using a temporary data structure to store the information before import. Most spreadsheets allow export of their data in *.csv format and for this reason they can be a very useful intermediate step between the source documentation and the final data storage. However using a spreadsheet almost always loses information stored in the document hierarchy unless a separate effort is made to recreate the hierarchy later (for this task refer to EMPRESS deliverable [D3.2.2], chapter 5.1).

For some requirements management tools it is easier to import the data in its original structure into the tool and then to manipulate it into the various abstraction layers. However is this done, the first aim must be to add information to the individual requirements. The usual way adding information is using attributes, which is covered in the next section.

5.1.5.2.4 Using attributes to categorize requirements

Before any restructuring the first thing that must be done is to uniquely identify the requirements. Adding a number to every requirement can do this. This identifier should be unique within the context of the whole project. It should remain attached to the requirement throughout the requirements life and should never be reused.

Other attributes that are attached to the requirements at this point are used to support the allocation of requirements to different abstraction layers. It is important to make a distinction between the various abstraction layers and to have a clear understanding of their functions.

The abstraction layers can for example be:

- Comments and explanations which are not requirements
- User Requirements
- System Requirements
- Subsystem and Equipment Requirements
- Interface Requirements
- Design Requirements
- Other requirements

Other sets of abstraction layers may be defined according to the technical domain in which the effort takes place. However, the above structure can in most cases be used as a baseline.

At this point it must be emphasized that two orders of processing the data are thinkable: either all information which does not contain requirement data is first identified, and only after that the requirements themselves, or vice versa. This depends somehow on the fact whether certain tools are used or no tools at all. For example, in DOORS the first way might be more appropriate since it is quite easy within this tool to identify for example headings. These normally are not containing requirements.

In order to understand the different abstraction layers as proposed above, the following explanations are given:

User Requirements Level:

User requirements are the first step towards defining the system. Unfortunately the word user implies the person who will be actually sitting and using the software or embedded system. A better term would be to use the word stakeholder, meaning anybody who can legitimately express requirements about the system to be built, including managers, maintainers, support engineers, and end users (see explanation provided in section 5.1.3.1). For the sake of simplicity we will continue with user requirements here.

User requirements should be expressed in the terminology of the problem domain, defining what users or customers want to do with the system. In other words, user requirements are defined from an operational point of view not in terms of system functionality or equipment. Many development approaches fail to distinguish between user and system requirements weakening the role of the user. User requirements and system requirements must be kept separate, with the former driving the latter. User requirements later form the basis for verification, in other words did we build the right system?

This level is also the logical place to define a coherent set of business requirements. Customer commitment and the system boundaries are often unclear at the start of the user requirements process. The business objectives act as the frame for the user requirements, defining the boundaries of any proposed system.

System Requirements Level:

System requirements explore the solution, but ideally avoid commitment to any specific design, in order not to restrict optimisation of the technical solution to be found by the developers. System requirements are aimed at showing what the system will do, but not how it will be done. The system requirements form a model of the system, acting as intermediate steps between the user requirements and the design. They are often couched in functional terms. System requirements later form the basis for verification, in other words did we build the system right?

It is not normally the users role to define system requirements, however it is quite normal in input documentation to find functional descriptions. These functional descriptions could be legitimate system requirements or they could indicate user requirements that should be expressed at the correct level. The authors of input documentation are not often experts in the systems to be defined, but they are experts in the problem that the system is designed to solve. The manipulation of input documentation defined in this process is designed to identify those functions the authors consider necessary. The next process would be to take those functions and define their real user requirements.

Subsystem and Equipment Requirements Level:

Subsystem level and the next lower levels of hardware and software may be covered by the input documentation as well, however, and hopefully, not to a high extent. The reason is that the more detailed technical information is present in the input requirements documents, the more the development and design optimisation of the system is limited.

The characteristics of subsystem and equipment information are that the reader needs from UR level down to Equipment or Interface level more and more knowledge of the technical domain.

The architectural design is not normally addressed in input documentation. However input documentation may define constraints upon the system to be built which define the architecture. Ideally input documentation should define the absolute minimum number of constraints on the architectural design of whatever level, and allow the designers as much design freedom as possible.

Interface Requirements:

In order to apply the elaborated rules, an effort has to be made by the requirements analyst to gather information on all ”neighbour systems” which have any link or relationship to the system under question.

For the Wiper System used as example for the study neighbour systems are the electrical system providing power, the data system providing data interfacing and control functions, the environmental conditions of the vehicle, etc. The requirements analyst has the task to find out which these systems could be, if not described in the input requirements documentation.

Design Requirements:

This type of requirements has not been covered within the available input requirements document (Annex A).

Design requirements are non-functional requirements. They concern performances, properties, and all “ilities” as for example reliability, maintainability etc. In the present reference document, no such notion has been used.

If however the stakeholders forward this kind of requirements, they must be carefully checked, whether there is a real need or obligation to specify them. This is necessary since design requirements limit the degrees of freedom for the designers and developers to optimise the implementation approach. However, there are often good reasons, why the customer brings in design requirements.

Other Requirements:

This type of requirements has not been covered within the available input requirements document (Annex A).

A requirement is an “Other Requirement”, if the information has been identified to be a requirement, however does not contain keywords of the above categories UR, SR, SSR, IR and DR. Due to this fact it has to be carefully checked by the requirements analyst, whether such requirements are justified and/or correct.

5.1.5.3 Applying given rules or finding other appropriate rules

The present report paragraph 5.1.4.3 provides a set of rules, which most probably will match a significant set of cases, be it for different domains, or for different authors and languages. A short check can be performed and if the rules do not provide sufficient coverage, it is then up to the requirements analyst to find out other, domain or author specific rules. This may be a considerable effort and a decision must be made, whether this is justified and efficient.

If alternative rules are looked for, the proposals made in chapter 5.1.4.5 should be first considered. Only when these are not successful, further ideas must be searched.

5.1.5.4 Achieving the text analyses

With the rules selected in the foregoing step the information elements of the Input Requirements Documents can be categorized according to the layers as identified in the chapter above. In order that the rules are usable, appropriate glossaries have to be prepared, for each abstraction level one (see example Table 5-6). This needs some detailed occupation with the technical domain in which the analysis takes place. It may be necessary to interview people working in that domain, or to read appropriate books.

Once the glossaries are produced, the process of identifying the given information elements must take place. This can be done via a commercial parser as available in DOORS, or by hand. For this purpose each information element must be analysed, which key words of the various glossaries are included. This leads to a table like Table 5-3, where the key words are spelled out for each line. According to rules, as the proposed rules 4.7 and 4.8, the categorization is then performed.

5.1.5.5 Extracting Requirements to the Requirements Model

Once the requirements have been annotated with the attributes showing the abstraction level to which it should be extracted, one can then filter to show subsets of the input document and copy that filtered subset into the requirements model. The requirements model itself will have a structure defined by the systems engineering standards in use. When we copy the subset of requirements into the model we should take care to copy the attributes and the unique identifier of the requirements as well.

If it is necessary to show trace ability to the original input documents then this should be done at this point, either by the use of another attribute giving the name of the source document, or by a linking mechanism. For details of this procedure refer to EMPRESS deliverable [D3.2.2] chapter 5.1.

The above activities represent the output interface to the next steps of EMPRESS requirements engineering, which are:

- Sub discipline Elicitation/Common Understanding
- Sub discipline Analysis/Verification-Validation.

For these activities see the EMPRESS process description.

The artefacts necessary for these further activities are requirements structured and stored in the form compliant to the information model of a project. In most cases this will be artefacts as produced by RM&E tools as Caliber, DOORS, Requisite Pro, SLATE, IrQA etc.

5.1.6 Different aspects and dependencies of the method

5.1.6.1 Language aspects of the method

5.1.6.1.1 Introduction to language dependencies

The present study has concentrated on a method, which has been able to be validated only for the English language. It is however of big interest to get an idea of whether it can be used for other languages, or even for any language.

For this purpose it is investigated, which dependence on language could appear with the rules found. Due to the limited degree of HOOD's EMPRESS involvement, only English, French, and German has been considered. From all rules documented above in section 5.1.4.3 only those rules have been investigated, which are relevant for the categorization. They all are related to the contents and meaning of words. Rules, which are only related to documentation formalism, as spacing, fonts etc. have not been considered. They are not dependent on the language under consideration here.

As relevant for many languages, there occurs often a mix of the language under discussion and the English language. This is due to the strong influence onto many industrial and scientific fields by the Anglo-American language as a worldwide language. In these cases English specific domain terms are used together with terms of the local language. Due to the reality of this situation, the rules to be found for input requirements categorization should take into account this fact.

5.1.6.1.2 Analysis of the defined rules w.r.t. language dependencies

This section checks the rules defined in paragraph 5.1.4.3 related to their sensitivity on different languages:

Rule 2.1	**A piece of text can be a heading, if no verb is used. Requirements text contain verbs (generally)**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. All three languages proceed according to the same rules to form headings.

Rule 2.6	**All texts between Headers can be either Requirements or Comments / Explanations**

Assessment: this rule is not dependent of any language.

Rule 2.7	**Each single piece of text, be it separated to the adjacent ones by full stop, TAB, line feed, or page feed, are objects which can either be a requirement or another text.**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent on a combination of them. The rules to form sentences using full stops are identical in all three languages. The use of special characters is common as well.

Rule 2.9	**Figures may contain requirement information or explanatory information; this is to be decided by quotations in the text and/or by legends or callouts**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. Figures are worldwide used as an engineering notation for explaining something. The used graphical notations are language independent.

Rule 2.8	**Tables contain multiple objects and are considered to be requirements**

Assessment: this rule is not dependent of the English, French, and German language, nor of a combination of them. A table is not a linguistic feature, but an information categorising and linking format, which can be applied in any language.

Rule 5.1	**A requirement is a UR if the text contains only nouns and verbs not belonging to a specific technical domain. They only are related to general physical parameters, to general actions and stati.**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. Nouns and verbs are being used in the same grammatical manner in all three languages.

Rule 5.2	**A requirement is a SR if the text contains nouns of low technical depth, not very specific to a technical domain.**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. The categorisation of a noun is a logical action and thus is not language dependent as such. There might however be different categories used in different languages, since the translations may not be exactly addressing the same contents.

This does not contradict the rule.

Rule 5.3 **A requirement is a SSR if the text contains technical, numerical, and physical details, domain related specialist terms, detailed stati and modes**

Assessment: this rule is independent of the English, French, and German language, nor dependent of a combination of them. The categorisation of a noun is a logical action and thus is not language dependent as such. There might however be different categories used in different languages, since the translations may not be exactly addressing the same contents. This does not contradict the rule.

Rule 5.4 **A requirement is an IR if nouns appear which do not belong to the system's product tree or functional content, however to external systems**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. The categorisation of a noun is a logical action and thus is not language dependent as such. There might however be different categories used in different languages, since the translations may not be exactly addressing the same contents. This does not contradict the rule.

Rule 5.5 A requirement is a DR if dedicated names of companies and product numbers or denominators, material qualities, design standards etc. are used.

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. The categorisation of a noun is a logical action and thus is not language dependent as such. There might however be different categories used in different languages, since the translations may not be exactly addressing the same contents. This does not contradict the rule.

Rule 5.6 A requirement is an oR if the information has been identified to be a requirement (rules 2.1-2.9), however does not contain keywords of categories UR, SR, SSR, and IR.

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. The categorisation of a noun is a logical action and thus is not language dependent as such. There might however be different categories used in different languages, since the translations may not be exactly addressing the same contents. This does not contradict the rule.

Rule 5.7: **In case the parsing result is ambiguous, the result of the lowest level takes precedence.**

Order of precedence (increasing priority): UR, SR, SSR, IR, DR

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them, however see the next section on compound words for details. This is a logical rule, which does not depend on the form of information presentation (language).

Rule 5.8 **If interface related key words are used in the information under analysis, and in addition key words of the categories UR, SR, or SSR are present, then the information is considered not to be an Interface Requirement, but of the category as defined by rule 4.7.**

Assessment: this rule is not dependent of the English, French, and German language, nor dependent of a combination of them. This is a logical rule, which does not depend on the form of information presentation (language).

The overall result of the assessment on language dependency is, that the rules defined are not dependent on the selected languages English, French, and German.

5.1.6.1.3 Compound words

Differences in different languages:

There is some attention to be paid to the existence of compound words. The technical language is evolving together with the evolution of technologies and specific domains. Nearly every day there are new words created, often composed by abbreviations or by assembling syllables of basic words. In some languages, as for example in French, the composition of compound words is not as simple as for example in the American English, where any creation of words is allowed and accepted by the domain participants. The French language uses conjunctions in order to produce complex expressions, for example "*commutatuer de collonne de direction*" for *steering column switch*. In German this word would be "*Lenksäulenschalter*". As can be seen, the French language introduces the word "de" in order to clarify, that the words "collonne", "direction", and "commutateur" are related to each other to form a dedicated device. In English, no such conjunction is necessary, however there are three single nouns necessary. In German, the two words "*Lenkung*", "*Säule*", and "*Schalter*" are just glued together to form one word, by deleting one syllable ("ung") and introducing an auxiliary character ("n") necessary for grammatical reasons. There are some tendencies in the English computer language to adopt a variation of the German system, by assembling nouns by using upper case characters: for example SteeringColumnSwitch, if we would like to remain in the automotive domain.

Language dependent impacts of compound words on glossaries production:

The above situation has an impact on the creation of glossaries in the different languages. The requirements analyst must take into consideration, how compound words are created in his language. The following Table 5-7 shows, which are the differences that must be considered by the process of requirements extraction from input documents. It also contains the situation w.r.t. Rule 5.7 of the method presented.

Language	User Requirement specific word	System Requirement specific word	Subsystem Requirement specific word
French	Direction	Collonne de direction	Commutateur de collonne de direction

English	Steering	Steering column	Steering column switch
German	---	---	Lenksäulenschalter

Language	UR	SR	SSR
French	Direction	Collonne de direction	Commutateur de collonne de direction
Rule 5.7:	X	X	X
English	Steering	Steering column	Steering column switch
Rule 5.7:	X	X	X
German	---	---	Lenksäulenschalter
Rule 5.7:			X

Table 5-7 Compound words processing

As can be seen, the English and French language necessitates that rule 5.7 be applied, whereas the German language does not need this in the chosen case, due to the creation of a specific new word for the *Steering Column Switch*. The result of the method applied will nevertheless be correct and identical in all three cases.

Combination of different languages:

In case of combinations of words from

- French and English (not very likely for the French nor for the English side), or
- French and German (quite unlikely for both sides), or
- English and German (quite likely for the German side),

The situation will most probably not change w.r.t. what is depicted in Table 6.7. However, this could be a topic for further research.

Impact on input requirements process:

The impact of the above situation onto the input requirements process is the following: The requirements analyst has to be aware of the fact, that compound words must not be forgotten whenever a text is analysed w.r.t keywords for the establishment of the glossaries. He must also know, that in his language he might be forced to identify the clusters of nouns, which form a compound word. If a cluster is the subject of a sentence, then he must not separate it into its elements and fill these elements into the glossary. He must in all those cases carry these clusters / compounds over to the glossary as a whole. For example, in English: “steering column” must remain together to form one single entry in the glossary for system requirements notions. It must not be separated into “steering” and “column” at this instance.

According to that, the following additional rules apply:

Rule 6.1	**If a requirement sentence (in English language) contains a cluster of nouns, it has to be analyzed whether this is a compound word, defining a very specific item.**

Rule 6.2	**The more elements the (English) compound word contains, the more it might belong to a lower level of abstraction (Subsystem or deeper).**

5.1.6.1.4 Results on language dependency of the method

- In summary it can be stated, that the applied rules are independent of the English, French,

Rule 6.3	**Each (English) compound word must be processed as a whole, and must not be divided into its elements.**

  and German language. Furthermore they are not dependent on a combination of them. This implies, that English, French, and German speaking engineers and computer specialists

Table 5-8 Glossary example in English, French, and German language

English				French				German			
UR	SR	SSR	IR	UR	SR	SSR	IR	UR	SR	SSR	IR
action	actuation	0	(car) speed	action	actionnement	0	vitesse (du vehicule)	Aktion	Einschalten	0	(Fahrzeug-) Geschwindig-keit
activation	delay	1	gear	activation	delai	1	vitesse	Aktivierung	Verzögerung	1	Gang
function	interval	2		fonction	interval	2		Funktion	Intervall	2	
position	motor	3		position	moteur	3		Stellung	Motor	3	
priority	pump	5		priorité	pompe	5		Priorität	Pumpe	5	
process	stop	15		processus	arret	15		Prozess	Halt	15	
rewipe	switch	<		re-essuyer	commutateur	<		Nachwischen	Schalter	<	
time	wiper	=		temps	essuye-glace	=		Zeit	Wischer	=	
wash		>		laver		>		Waschen		>	
wipe		cam		essuyer		came		Wischen		Nocke	
		clip				connexion				Klemme	
		high				haut				hoch	
		km				km				km	
		low				bas				niedrig	
		mode				mode				Modus	
		OFF				circuit coupé				aus(geschaltet)	
		ON				circuit fermé				ein(geschaltet)	
		one				un				ein	
		relay				relais				Relais	
		single				seul				einzeln	
		speed				vitesse				Geschwindigkeit	
		steering column switch				commutateur de collonne de direction				Lenkradschalter	
		three				trois				drei	
		tip-switch				commutateur touche				Tippschalter, Taster	
		twice				deux fois				zweimal	
		two				deux				zwei	

can utilize the method proposed. There is a high probability that a lot more European languages are able to apply the set of rules found as well. Since grammar and words stem from some common roots (Latin, etc.), they do not differ much in their application for technical documentations.

As a consequence, it does not matter whether these languages use a mix of words from different European languages. This is most helpful, since it is easy to handle, and understandable by any engineer. No linguistic skills are necessary, or any specific knowledge on grammar and spelling.

Table 5-8 shows for the three chosen languages (using the reference document “Wiper System”), that all notions and words are corresponding to each other and can be handled by the method in an identical way.

Each English word has a corresponding German or French word. Therefore the glossaries, which would be produced in the cases of a German or French input requirements document would be very comparable to the one produced for the English wiper document.

However, here is one topic to be considered: all compound words need specific attention, in order not to break them up into their elements, which would not lead the glossaries to be successful in the sense of the present study.

5.1.6.2 Tools aspects of the method

The input requirements process described in subchapter 5.1 has performed an analysis of an EMPRESS internal example document in order to define the rules, techniques, and methods to derive structured requirements from a set of unstructured input requirements information. For the purpose of the study MS-WORD and MS-EXCEL have been used as initial tools. Later on, DOORS has been introduced and discussed as a tool, which is able to provide a considerable improvement to the process, in that it allows the requirements analyst to perform certain tasks semi-automatically, without needing to program a lot of features in Visual Basic Applications (VBA) for WORD and EXCEL, or in a standard programming language.

There are different actions within the input requirement method, which need not to be done by hand. In all these cases it is worthwhile considering the use of tools. Normally, input requirements documents are available in some form of a text file, be it as a *.txt, *.doc, *.wp file, or other word processor files. On the basis of these files, the actions as listed further below can be somehow performed automatically, or at minimum supported by built-in tools of the word processor used. In addition, spreadsheet functions can be useful.

Since companies often use off-the-shelf data bank programs for requirements documentation (ACCESS or similar), the related tools can be used as well. Normally, special tools for requirements management like DOORS, RequisitePro, Caliber, etc. are data base system and provide a lot of useful functions to support the input requirements process discussed here.

The following list identifies the main functions, which HOOD needed in using the method for the reference document (wiper system input requirements). Whether they are used within the word processor, the spreadsheet program, or the data bank system, is left to the decision of the analyst, since a lot depends on which application program somebody is used to.

- Operations for Glossary production
    - Filtering of a list for different purposes and using different filtering variables, combined with Boolean rules
    - Find and replace functions, case sensitive; special characters must be possible
    - Transformation of tables into structured text

- General Operations:
  - Possibility to save in an appropriate format (*.txt, *csv, etc.) for export to other tools
  - Possibility to import different text formats into the used tools

In any case, if the input requirements documentation is very voluminous, it is worthwhile to use specific menu functions available with the office application programs. They allow automation of specific actions necessary for the method and thus support the whole process in a useful way. It is also possible to achieve a high amount of simplification, if macro programming is used. Both activities, however, necessitate a certain amount of work. Therefore it is not appropriate in all cases to decide for a programming effort. The best solution might be to use the standard office tools and a requirements management tool (if available), with their evident menu functions. It is however a question of whether the combination of application programs available at an organisation offers a sufficient support, since it is not a priori evident, that these programs are complementary to the required overall support.

During the study, DOORS has been examined w.r.t. its support of the process, and its functions have been studied as to their usability for the method. The following functions of DOORS are able to support the analyst's work considerably:

- Import from MS-WORD covers rules 2.1 through 2.8
- The imported files can be put under strict read-only access
- The various input requirements documents can be imported into DOORS as read-only files, called "Descriptive Modules"
- The categories for the requirements can be defined as DOORS attributes, as well as any other information representing a requirement attribute
- The linking between the input requirements and the categorised requirements can be performed immediately, if the input documents are imported as Descriptive Modules, and the categorised requirements are consolidated onto formal modules of the wished levels of abstraction
- DOORS allows a comfortable and complex filtering of requirements, associated with the creation of views. These views can be the categorised documents originally aimed for in the project's information model
- The glossaries, if elaborated in EXCEL, can be imported into DOORS as formal modules
- DOORS allows mapping these glossaries to the requirement to be categorised. This however needs programming effort using the DOORS specific programming language DXL.

- In summary, it is recommendable to use DOORS for the implementation of the rules found. It has however to be pointed out, that other requirements management tools can be equally useful for this work. This should be examined in another research study, or by the company itself, which intends to apply the proposed rules and procedures.

5.1.6.3 Embedded systems aspects of the method

The method presented is of a very generic character. It accepts words of any specific domain, be it *embedded software systems* or any other technical area. Domain expertise has however to be implemented in the glossaries needed for the method. But once more, this is not specific to the domain of *embedded systems*. The requirements analyst of an *embedded system* will have to perform the same tasks for applying the method, than any other requirements analyst.

5.1.6.4 Evolutionary aspects of the method

5.1.6.4.1 Starting point

The support of an evolutionary development approach is an important issue in the EMPRESS study. In order to assess the ability of the input requirements process described in subchapter 6.1 to be compatible with an evolutionary development process, this process is once more shown in Figure 5-9 and shortly explained:

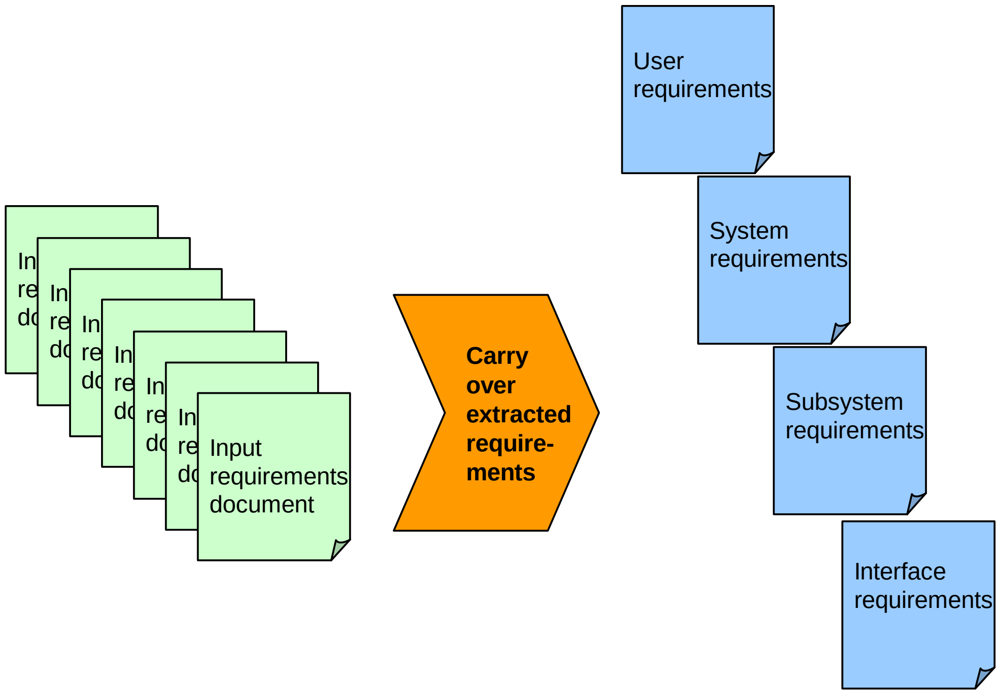


Figure 5-9 principal input requirements process schematic

From a set of input requirements documents originated by different stakeholders and of different form and contents, all information is identified which corresponds to requirements. All other information is not directly used and only serves for a better understanding of the requirements. All identified requirements information is then carried over to the information model of the project, which might consist of documents as:

- User requirements
- System requirements
- Subsystem requirements
- Interface requirements.

Other information structures are possible, and the method is not sensitive for these potential differences.

In a waterfall or incremental development process the above shown input requirements process would be performed just once, namely at the beginning of the project. For an evolutionary process this is no more the case. There, small development cycles with requirements definition, design, implementation (coding and integration), and test are performed. The evolutionary concept is hereby realised by continuously evolving requirements (see Figure 5-10). It normally starts with high priority requirements, and proceeds with every cycle to the lower priority requirements. This is of concern for the method described in this chapter 5.1, and is explained in the next section.

At each of the cycles, after a successful test, there is a product available, for example an early software release. It serves as an input for the next, evolutionary cycle, together with a delta set of further requirements, which might come from the earlier input requirements documents, and/or from newly issued ones.

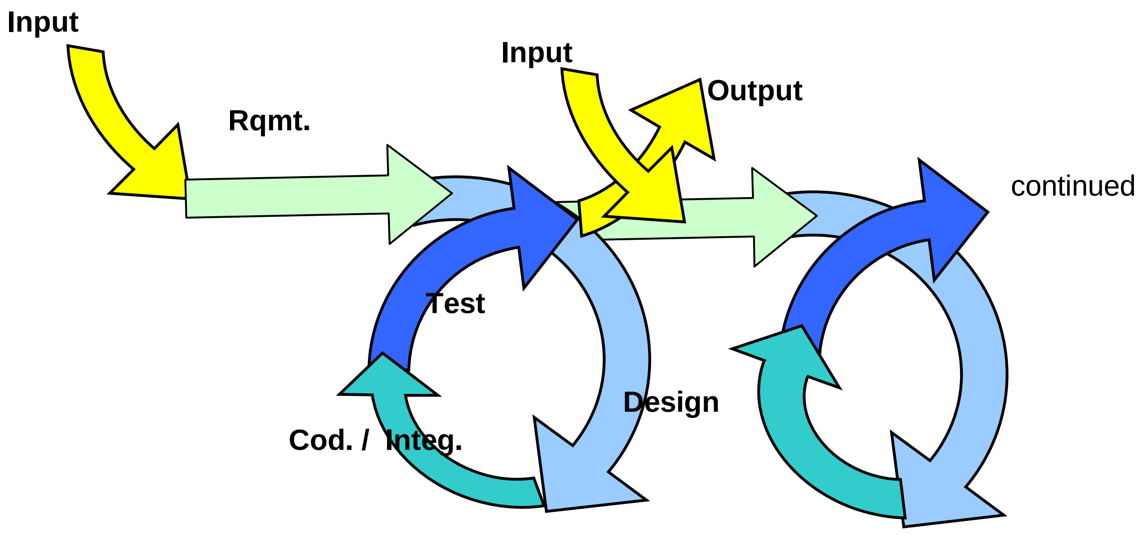


Figure 5-10 evolutionary cycles of a development process

5.1.6.4.2 Evolutionary development process and input requirements process merger

In order to merge both processes, it is necessary to analyse the needs of the evolutionary process w.r.t. requirements processing. As already mentioned above, the evolutionary process is evolutionary in the sense, that it considers input requirements step by step, for example according to their importance. This importance is defined by giving each single requirement available at the project beginning a certain priority. This is in general to be performed by the project manager, and the requirements analyst. The latter will have derived his initial requirements by analysing the input requirements from various stakeholders, according to the input requirements process described in this study.

After the first basic cycle of development performed, the first evolutionary step will follow. For that purpose, the requirements available at that point in time will have to be analysed again, and screened for requirements with the highest priority now. This may be a set of requirements, which had not been considered at the first basic cycle, and in addition a set of new requirements of high priority, stemming from a new set of input requirements having arrived meanwhile from various stakeholders.

This loop can be repeated as often as necessary and appropriate for the development under question. The overall process is shown in

Figure 5-11.

5.1.6.4.3 Conclusions on evolutionary aspects

The input requirements process proposed is entirely compatible with an evolutionary process. It can be repeated at each evolutionary cycle of the overall development. The first and basic cycle may however be the one necessitating the highest amount of effort, if at that time already the majority of requirements is available.

Product lines, which to a certain extent are comparable to an evolutionary approach, can be covered by the proposed method as well.

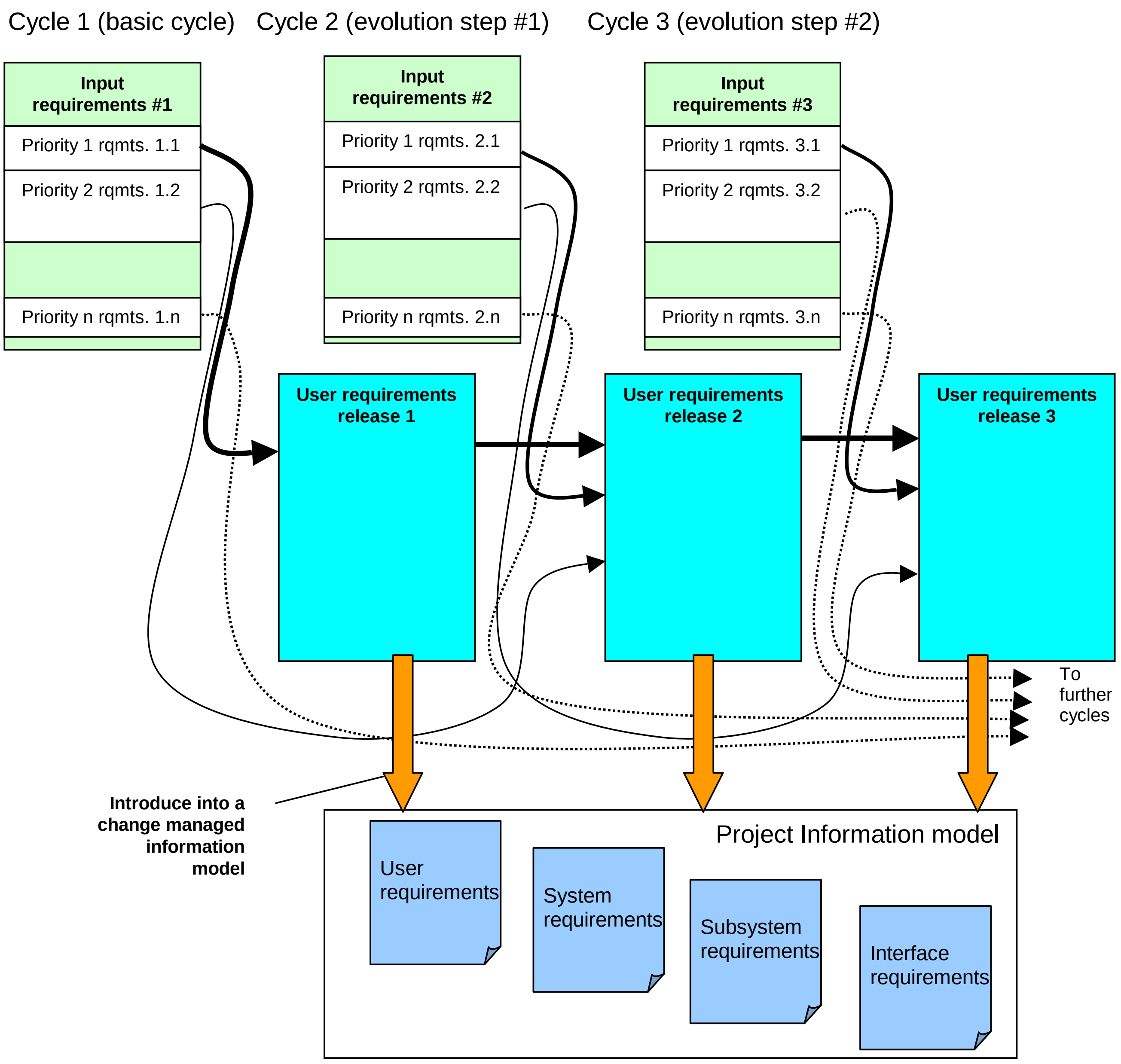


Figure 5-11 evolutionary input requirements process

5.1.6.5 Types of requirements and the method

The input requirements method has so far not made any distinction between functional and non-functional requirements. The purpose of this section is to show, what principal difference is to be considered for both types within the input requirements method and process, and whether there is a difference at all.

The method is based on the principle, that for each input requirement the related function or item is named within the requirements sentence, and in general is related to an abstraction level according to the glossary of section 5.1.4.3.8. For example, in the Wiper System reference document, the following sentence is an appropriate example of a functional requirement (InR58):

> “The wash function continues as long as the switch is pressed down.”

In this sentence, the notion “wash function” represents the subject and the abstraction level “system”.

The same principle applies for a non-functional requirement, as for example a design requirement (DR). Unfortunately, the reference document does not contain any non- functional requirement. However, a related example fitting the wiper system requirements document might be:

> “The wiper control system software shall be implemented using the programming language ABC. “

This sentence as well specifies, to what subject the design constraint belongs: it is related to the notion “wiper control system software”. As such it is possible to categorise this requirement as to be a subsystem related requirement. The condition for this is that the requirements analyst has decided for defining this item being part of the subsystems level of abstraction and its related glossary. However, on the other hand the requirement contains the notion “programming language ABC”, which would be part of the design requirements glossary, since it has no basic functional relevance. In this case the requirement would be classified to be a system requirement according to rule 5.2, and to be a design requirement according to rule 5.5. Although two rules are correctly applicable in this case which represents an ambiguity, rule 5.7 solves the problem.

As can be seen it is quite probable, that non-functional requirements can be covered by the method described in the same way as functional requirements. Due to the fact, that the reference document used (wiper system) does not contain any non-functional requirement, this is however a hypothesis and should be proven by another case study with appropriate input requirements documents containing non-functional requirements.

5.1.6.6 Linking aspects of the method

The input requirements process as defined in the present study yields a complete list of requirements extracted from unstructured and mixed information input documents. These are later on carried over into the information model of the project, which means into the related requirements documents or requirements database.

In order not to loose any information based on the relationship between the source of a requirement and an information model (database or requirements document), linking must be performed between the input requirements and the derived formal requirements. There are a set of methods and activities to be considered, which are elaborated in another part of the EMPRESS project, namely in the requirements management section of the project. Therefore the EMPRESS deliverable document [D3.2.2] provides the related input requirements linking study (chapter 5.1, provided by HOOD-GmbH).

5.1.7 Study Results, summary and outlook

The report presents the starting point, the performed work and the results of a study, which discusses methods to categorize requirements issued by the customer of a system to be developed. These requirements are normally covered in a set of diverse documents of different format and character, possibly originated by different stakeholders. The requirements themselves are generally not written according to strict rules most appropriate for requirements writing, however consist often of poorly structured and worded information.

In order to study the above problem, a typical technical specification document from automotive industry has been taken as the object of an initial research study. The texts have been analysed, and rules of identifying and categorizing the requirements have been elaborated.

The results of this effort are:

- It was possible to find general rules to identify requirements for the different abstraction layers selected in this case study
- The results yield a high success rate, however the parsing of requirements of the lower levels (here: Subsystem Requirements) has been the most failure prone area
- Tables of requirement information can be analysed with the rules found, too
- There is a certain effort necessary to establish glossaries for the different abstraction layers. They are dependent on the technical domain of the system to be developed. The glossary for the Subsystem Requirements and Design Requirements layers are the most voluminous ones
- The method is not sensitive whether the specification language is English, French, or German
- The method is not sensitive to a mixture of these three languages (especially German and English)
- The method is useful for embedded systems, however not limited to this domain
- The method supports an evolutionary development approach, which is a main focus of EMPRESS. As such it is able also to support product line approaches.
- The method is not sensitive to different categories of requirements, such as functional and non-functional requirements.

The benefit of the application of the defined parsing rules is to facilitate the work to be done in those cases, where a multitude of input documents with up to thousands of sentences and hundreds of tables are delivered by the customer and must be digested by the contractor. It most probably will save a considerable amount of time to apply an automatic parser to this problem, which would be based on the rules found. Although such a tool will, due to uncertainties not resolved with the defined rules, not yield a 100% coverage of categorization, it is nevertheless helpful for the requirements analyst to have some major help in his time consuming work. In the case that the method will be used to perform the work by hand without any tools such as mentioned above, there will nevertheless be a considerable benefit, since the analyst will have a guideline for his work and needs not think too much about how he could solve his problem.

It is to be mentioned that there will in general efforts be necessary to customize the rules and necessary glossaries of nouns to the technical domain which the system under development is part of. This field however has to be worked on in the future, if it is possible in later ITEA projects.

Further findings of the study have been, that, in order to validate the rules found, a larger set of diverse input documents has to be analysed. This is especially true in the domain of software engineering for embedded real time systems, which was not possible to be performed with the reference document available during the EMPRESS project. Additional work should also be carried out using input documents from various authors and stakeholders, and in different languages. These extensions could not be performed in the present study due to the limitations imposed by the initial EMPRESS planning. However, the HOOD-GmbH performed work on the method demonstration, as documented in Work Package 5 of the EMPRESS study, covers some of these aspects, using a set of Input Requirements Documents from different stakeholders. The results of that demonstration have confirmed the assumptions, findings, and rules of the present study.

A further future effort could be to develop a text parser performing the task of the found rules, or at minimum to evaluate the performance of parsers as presently available on the market. In addition one could find out, whether the adaptation of known rules or the finding of new rules, and the subsequent automatic parsing are more efficient than the analysis “by hand” performed by an experienced requirements analyst. Parts of these possibilities are covered by HOOD-GmbH in WP 5 as well.

It is to be pointed out, that the aspects of linking the input requirements information to derived, categorised requirements, is covered in the EMPRESS deliverable [D3.2.2] ch. 5.1.

Annexes

Annex A: Motor Vehicle Wiper System Input Requirements Document

see accompanying document [WS

5.2 A Conceptual Model describing the Evolution of Textual Information into a Product Line

5.2.1 Introduction

In this section we will describe a contribution by Fraunhofer IESE. We describe the general concepts of product line engineering and especially requirements engineering for product lines. We describe an approach here to extend use cases to be able to model common and variable requirements on a product line and describe a conceptual model that will support requirements elicitation for product lines based on user documentation.

5.2.1.1 Relation to the Empress Process

This work describes an elicitation approach. So, the work described in this section is strongly related to Requirements -> RE-> Elicitation. Within this activity, the sub activity scoping and collection are covered. Another activity that is related is Requirements -> RE-> Analysis and within this activity the sub activity classification.

5.2.1.2 Evolution and Product Lines

Product line engineering can be described as a technology providing methods to plan, control, and improve a reuse infrastructure for developing a family of similar products instead of developing single products separately. This reuse infrastructure manages commonality and controls the variability of the different products.

Examples for product line approaches are PuLSE [Bay99a], Fast [Wei99] and the SEI Product Line Practice Initiative [Cle01]. The goal of product line engineering is to achieve planned domain-specific reuse by building a family of applications. Distinct from single system software development there are two life cycles, domain engineering and application engineering. In domain engineering the reusable asset base is built and in application engineering this asset base is used to build up the planned products (cf. Figure 5-12).

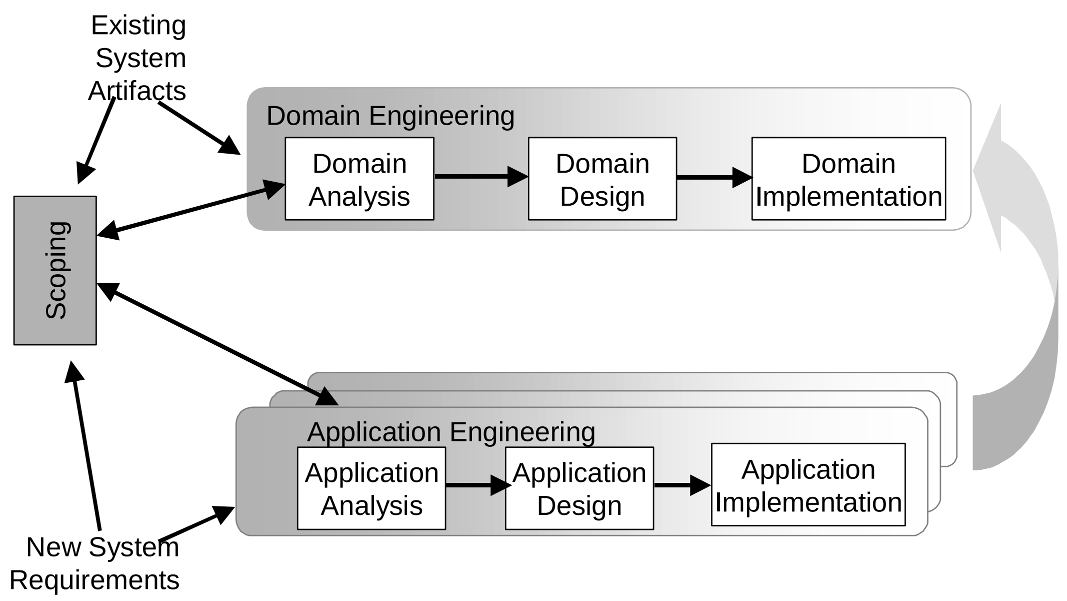


Figure 5-12 Product line engineering: A two lifecycle approach

One benefit of the product line approach is the idea that all members of the product line are based on a single set of assets. Thereby, the maintenance effort is reduced to this single asset base. Hence, existing products must always follow the evolution of the asset base. This is done by instantiating the changed domain model while reusing the existing resolution of the domain decision model.

Evolution happens in different ways in the product line:

- **Evolution into a Product Line**
  Systems developed at an earlier point in time have to be integrated within the product line. The research questions that have to be answered here are: How can evolution into a product line happen? How can systems be integrated? Which information from legacy systems do we need and which information normally is there in the context of embedded systems?
- **Planned evolution within a Product Line**
  Systems within the product line are derived from the common platform in application engineering. The research questions that have to be answered here are: How does the evolution within a product line look like? Which means does product line engineering provide to support changes?
- **Unplanned evolution within a Product Line**
  The systems derived from the platform evolve after they are instantiated from the product line and the platform itself evolves also because of new requirements and technological changes. The research questions that have to be answered here are: How can unplanned changes be incorporated into a product line? How can product lines be maintained?

In our work we will focus on the first area, evolution into a product line. When creating a product line or incrementally integrating products, it is important to use existing information on legacy products. For creating a domain model however so far no solutions exist for the systematic integration of legacy information (like user documentations or specifications).

5.2.1.3 Research Questions and Industry Needs

The work we describe here will contribute to the following research questions and industry needs mentioned in deliverable D1.1 part 3:

Research Questions:

- How can evolution into a product line happen?
- How can existing systems and their documentation be integrated?
- How can textual information serve as a valuable information source for product line modelling and scoping?
- How can experts be relieved from reading all those documents when integrating the information into a product line?

Industry Needs:

- Definition of a well defined elicitation process
- Contribution to systematic requirements recycling and reuse
- Definition of a use case model
- Improvement of the interplay between textual requirements and models (by defining possible transformations)

- Partial contribution to auto checking and enforcement of requirements structure
- Tools to automatically generate requirements

5.2.2 Requirements Engineering for Product Lines

5.2.2.1 Overview

Product line engineering stands on a middle ground between specific assets (single system development, one time use) and general assets (general use libraries or components, applicable anywhere; generator-based approaches). The idea is to define a product line, which captures the intended scope of reuse. The products included in a product line are determined to have sufficiently common characteristics to make it more efficient to study the commonalities and variability once for all products of the product line and to build reusable assets, than to study and build all the products separately. Product line engineering approaches define how to leverage the commonalities and create reusable assets that increase the efficiency of developing the products [Bay99b].

Domain analysis or domain modelling, is requirements engineering for product lines. Domain analysis methods provide processes for eliciting and structuring the requirements of a domain, or product line. The results are captured in a domain model. A domain model must capture both the common characteristics of the products and their variations. The domain model is the basis for creating other reusable assets like a domain specific language or a component-based architecture. For a domain analysis method to be applicable it must be appropriate and it must provide enough guidance so that it can be carried out. As in other areas of software development, the context for each domain analysis application varies, and methods that are appropriate in one context will not be in others. This fact is especially important for domain analysis because of the compound effects of inappropriate models over multiple products and over the whole lifecycle. Therefore, a generally applicable domain analysis method should be customisable to the context of the application.

The special properties, which are specific to requirements engineering for product lines, are:

- Commonality and Variability
  When doing domain analysis the properties of several products have to be modelled at once. As the planned products that are analysed during domain analysis differ in their features and in their functional and non-functional requirements, the commonalities and variability between those products have to be captured and adequately modelled.
- Instantiation Support
  As several products are modelled in one domain model it must be clear, which part of the model or which requirement belongs to which product. In order to have an application specific view on the product the instantiation of the generic and variable model for several products has to be supported.
- Decision Modelling
  To get this instantiation support, the decisions that have be made have also to be captured in a separate model. This model collects and abstracts the information on which requirement is instantiated in which product and supports the instantiation.
- Traceability
  Providing traceability from the requirements to the product and from the requirements to architecture, implementation, and tests is very important in product line engineering. As a product line spans over several products and several releases of the products it has to be ensured that those two dimensions of

traceability (traceability through products and through lifecycles) are provided.

- Evolution
  Product Lines are a means to cope with evolution. With product lines evolution in space (the space of the planned products) is controlled. When doing domain analysis on a portfolio of planned products evolutionary aspects are integrated and the evolution within the product portfolio is captured through commonality and variability modelling.

5.2.2.2 Product Line Requirements Elicitation from Documents

It is not reasonable to build a product line from scratch without having a broad area of expertise in the domain of the product line [Cle01]. The ability to think, design and develop in a certain generic way and finding variability within a domain and a set of envisioned systems needs a certain amount of domain understanding. The concept of domain understanding is described in the framework for software product line practice [Cle01] in the following way: "Domains are areas of expertise that can be applied to the creation of a system or set of systems. Domain knowledge is characterized by a set of concepts and terminology understood by practitioners in that area of expertise. It also includes an understanding of recurring problems and known solutions within the domain".

This domain understanding can be increased by analysing existing systems. The integration of these systems increases the amount of reuse by reusing all possible legacy assets and the traceability and completeness by establishing links to domain specific and organization specific work, which has already been done. Starting a product line engineering approach in an organization typically requires a complete and radical change of the environment, which inherits risks that cannot be foreseen and are difficult to be estimated for the organization. To be able to estimate and prevent those risks, domain understanding is a key factor for success.

Normally there are legacy systems with many existing legacy assets in different forms like code, documentation etc. which cannot be ignored when starting to develop a product line. Those assets are a rich source of knowledge and offer cheap available information, which already exists and is specific to the domain.

The integration of those existing systems into a planned and to be built product line can happen on different levels [Joh01]:

- Analysis and integration of code
- Analysis and integration of the architecture
- Reuse and integration of knowledge and expertise

Each one of these levels has meaningful information for the product line that is to be built or already existing. As already described in deliverable D1.1 part 3 there are several approaches for domain analysis and domain modelling. But in most of these approaches, the integration of legacy systems into the domain analysis phase is not described in depth. ODM [ODM96] is an example where software systems are described as a source of legacy knowledge, which can be integrated into the product line by reverse engineering. But there is lack of systematic support for integration of all possible assets into the lifecycle phases of a product line.

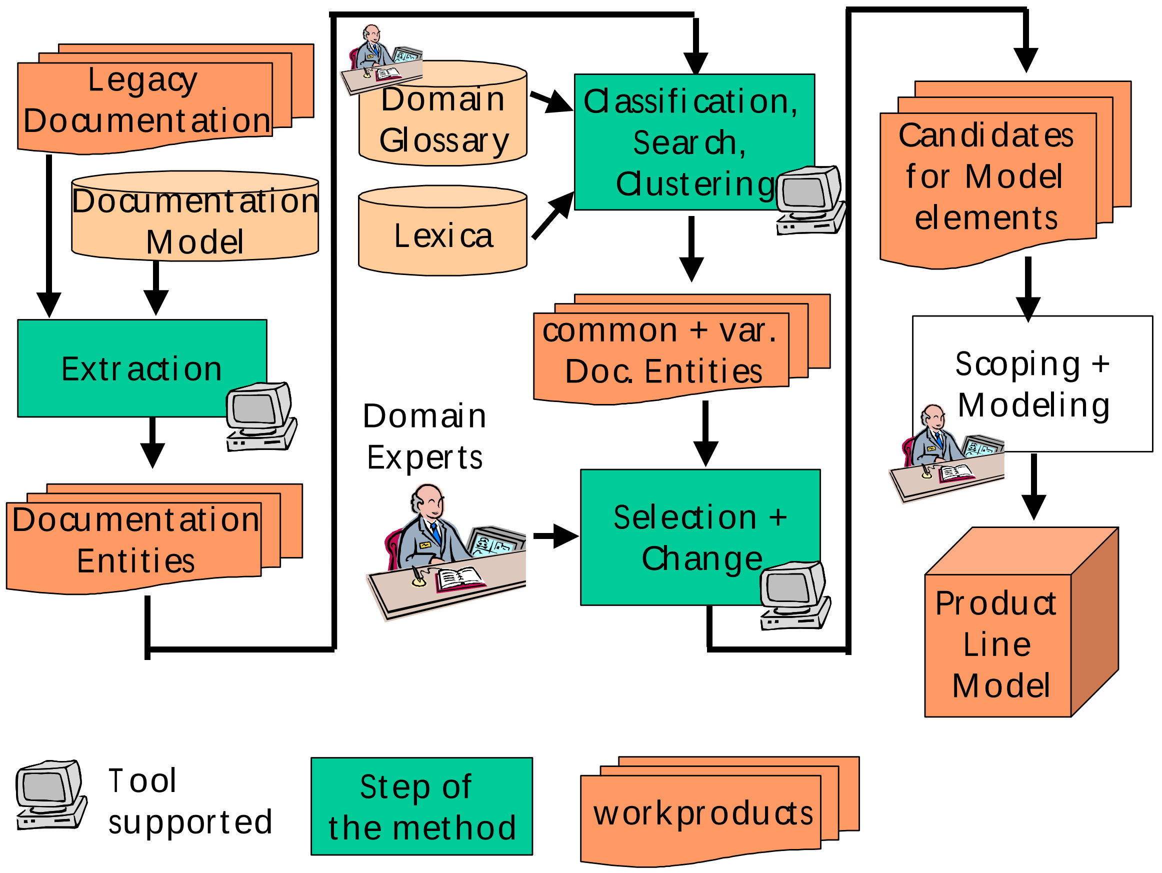


Figure 5-13 An Elicitation Process

The steps of a method for the integration of legacy documentation assets into a product line are described in Figure 5-13. Legacy documentation of all kinds (user documents, requirements specifications, other documents), which comes from different systems, serves as information source for the product line elicitation method. A documentation model that describes the elements of the legacy documents, the product line model elements and the transformation between them is also an input. First, the documents are extracted in order to get small documentation entities of one phrase or one sentence or paragraph. Then the documentation entities are classified and clustered in order to find entities that are common to all documents, that are variable between the different systems, that fit into a certain subdomain, or that relate to one feature. After this automatizable clustering and classification is done, experts can look at the classified documentation entities and decide if the commonalities and variabilities that were found should be commonalities and variabilities in the planned systems of the product line also. The outputs of the process are the documentation entities that fit into the product line model for the envisioned product. With these documentation entities as a basis, product line modelling can be done easier and the results are more correct and complete (than with modelling “from scratch”).

5.2.3 Product Line Modelling

In the above chapter an overview is given on a process for the elicitation of product line information. The outputs of this process are normalized documentation entities that should serve as input for product line modelling. In order to be suitable for product line modelling, these documentation entities have to fit to the selected modelling formalism. Different

modelling formalisms (including ODM, FODA….) were already described in Deliverable 1.3 part 3 and will not be described again in detail. A comprehensive introduction on product line modelling can also be found in “Product Line Analysis: A Practical Introduction”, CMU/SEI-2001-TR-001 (see [Cha01]).
Feature Modelling is one of the standard formalisms for Product Line Modelling (see [Kan90], [Gri98] and [Cza00]). As it was often described in literature we also do not go into detail here. Design Spaces are another example formalism used for Product Line Modelling [Bau00]. As UML and Use Cases have widespread use in modelling of requirements and in design, use cases should be considered as modelling formalism for product lines. There is no standard mechanism for extending use cases for product line modelling. Therefore in the following section we propose a mechanism for extending use cases in order to express variability. This work has been described in two publications ([Joh02a] and [Joh02b]).

5.2.3.1.1 Introduction to Product Line Modelling with Use Cases

Use cases are used for single system requirements engineering to capture requirements from a customer/user point of view. When utilizing use cases for product line modelling they have to be extended with a variability mechanism. Stereotypes can be used as the variability mechanism for use case diagrams and textual use cases. This early and explicit variability in the product line lifecycle supports the domain experts in establishing a variability mindset and supports explicit instantiation during application analysis.

Use cases are a widely accepted means to support domain understanding and to find, and document user requirements but there is no generally accepted formalism that integrates variability modelling with use cases. Expressing variability in the use cases has benefits in the following ways:

- Seeing variability in the use-cases helps all involved roles in establishing a variability and product line mindset and in getting a better domain understanding.
- Explicit variability in use cases supports the instantiation and derivation of exact models in application engineering.
- In market development that is not customer-oriented, variable use cases are a good means of communicating the possibilities of the possible products between marketing and requirements engineers and product line engineers.

There are some approaches on how to extend use cases with variability and how to support reuse and genericity in use cases. Biddle et.al. [Bid02] suggest using patterns in use cases to express variability and to organize the use cases in a repository. However, they do not introduce variability. Gomaa [Gom00] introduced in his method the stereotypes <<kernel>> and <<optional>> for use cases and other UML model elements for modelling families of systems. In his approach he focuses on the integration of features and use cases but does not say anything about how textual use cases should be represented. Jacobson et.al. [Jac97] discuss how the text in use cases may involve variation points that can form the basis for a hierarchy of use cases from more abstract to more specific. They introduce “variation points” (notated as dots in use cases including a short description of the variation) into use case diagrams. They also use variation points in textual use cases (notated as highlighted text in curly brackets) to describe different ways of performing actions within a use case and also discuss the use of include and extend relationships. They do not say anything about how variant or generic use cases can be instantiated. The approach we propose here is derived from the KobrA approach [Atk01], which introduces variation points in the UML including use cases and supports the instantiation of generic models.

5.2.3.2 Single System Requirements Engineering with Use Cases

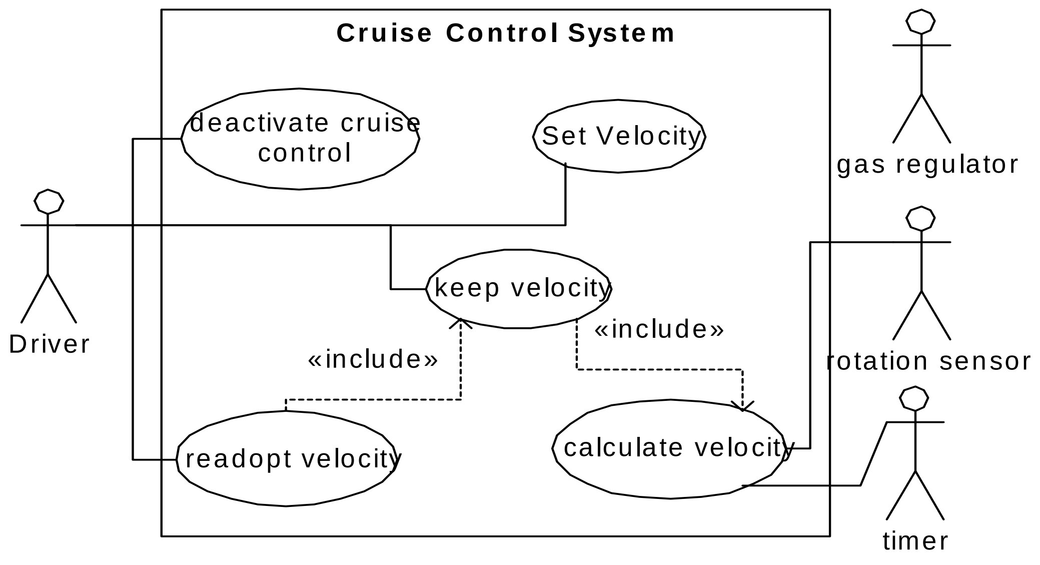


Figure 5-14 Use Case Diagram

Requirements Engineering for single systems has been done for a long time. Formalisms that have been used for a long time are e.g. textual requirements, controlled languages and formal or semi-formal specification languages like SDL or Z. For some years use cases [Jac92] have been a rather often used means to understand, specify, and analyze user requirements. Use cases can document the requirements on a system from a user's point of view. Use cases focus on functional requirements, they normally do not deal in depth with non-functional requirements, with interfaces and data formats. But use cases make it easier to understand what the system does and give a good means of communication about the system. We will describe an approach to extend use cases with variability and will illustrate it on the example "cruise control system". A cruise control system supports the driver of the car in keeping a constant velocity.

Use-Case Diagrams

A use case describes how a user uses the system. Use cases are used during the analysis phase to identify and partition system functionality. A use case describes the actions of an actor when following a certain task while interacting with the system to be described. A use case diagram includes the actors, the system, and the use cases themselves. The set of functionality of a given system is determined through the study of the functional requirements of each actor, expressed in the use cases in the form of common interactions. So a use-case diagram in UML 1.4 consists of [OMG01]:

- The system
- The use cases within the system
- The actors outside the system
- Relationships between actors and use-cases:

  associations, generalization, include, and extend

Associations denote the participation of an actor in a use case. A generalization relation means that there is a specialization of one use case or actor to another. An extend relationship indicates that an instance of a use case may be augmented by the behaviour specified by another use case and the include relationship indicates that an instance of a use case will contain the behaviour of another use case.

Figure 5-14 shows an example use case diagram for a cruise control system. The driver activates the cruise control system by choosing “set velocity”. He can also tell the system to “keep velocity” with help of the gas regulator if the velocity has already been set. He can also “readopt velocity” which will bring the car to the fixed speed (e.g. after braking) and then continue keeping the velocity. In order to keep the velocity the system has to “calculate velocity” with different sensors.

Textual Use Cases

There is no standardized form for the content of a use case itself. The standard describes the graphical representation and the semantics of use case diagrams only. Use cases are fundamentally a text form although they can be written using flow charts, sequence charts or petri nets [Coc01]. Use cases serve as a means of communication from one person to another, often among persons with no training in UML or software development like end users or marketing staff. So writing use cases in simple text is usually a good choice. There is no general agreement on the attributes use cases should have and on the level of description of the use cases. Figure 5-15 shows an example of a textual use case in the cruise control domain, which describes the use case “keep velocity”. The template used is a modification of the template suggested by Alistair Cockburn (see [Coc01]). The use case is described with its actors, the triggers, which means the actors that can activate the use cases. The input and output of the use case are described and the postconditions and a success guarantee (what the user wants from the use case) and a minimal guarantee (what should in any case not go wrong) are given. The main part of the use case is the main success scenario, which describes what the use case actually does.

5.2.3.3 Product Line Use Cases

In this section, the use case approach is extended to product families, that is, use cases do not longer describe the actions of an actor when following a certain task while interacting with a particular system only but summarize and integrate use cases describing analogous tasks for different products in a family into combined artefacts, product-line use cases. By making

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
Actors: driver, gas regulator
Trigger: actor driver
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Sucess Scenario:

```
    <keep velocity> is selected by actor driver
    get vactual, vtarget (<Calculate Velocity>)
    - compare vactual and vtarget
    If  vactual < vtarget : gas regulator increase velocity
    - restart <keep velocity>
    If  vactual > vtarget : gas regulator decrease velocity
    - restart <keep velocity>
    else restart <keep velocity>
```

Figure 5-15 A Textual Use Case

explicit this variability in the use cases, this variability can be communicated to all relevant groups (requirements engineers, product managers, management staff...), a variability mindset is established and a controlled derivation of application models can be supported.

Product Line Concepts

From an abstract point of view it is the concurrent consideration, planning, and comparison of similar systems that distinguishes product line engineering from single-system development. The intention is to systematically exploit common system characteristics and to share development and maintenance effort.

In order to do so, the common and the varying aspects of the systems must be considered throughout all life-cycle stages and integrated into a common infrastructure that is the main focus of maintenance activities. Commonalities and variability are equally important: commonalities define the skeleton of systems in the product line, and variability bounds the space of required and anticipated variations of the common skeleton.

Applied to use cases, these concepts produce use cases that have a common story that is valid for all members of a system family with variation points that explicitly capture which actions are optional or alternatives. Of course, a use case as a whole may be optional, as well as use cases under the same label may be realized totally different for some products. The common parts are modelled as all parts in a single-system context, to model variation, additional means are required. The variant use cases are instantiated during application engineering. The instantiation process is guided by a decision model, which captures the motivation and interdependencies of variation points, and produces use case artefacts as used in a single-system context (see previous section). In the following subsections, one way of modelling variation in use-case diagrams and textual use case descriptions is introduced.

Generic Use Case Diagrams

In use case diagrams, any model element may potentially be variant in a product-line context. An actor is variant, for example, if a certain user class is not supported by a product. A use case is variant if it is not supported by some products in the family. Those variants can be alternatives, optional elements or value ranges for certain elements. Whether it is an optional use case or whether it is an alternative to another use case is captured outside of the use-case diagram in a decision model. This is done simply because this information would overload the use-case diagram, make it less readable, and thus less useful. A generic use case diagram for our cruise-control example is depicted in Figure 5-16. There, an optional distance regulator has been added, that is, four additional (and variant) use case have to be modelled by using the stereotype <<variant>>, as well as an additional actor, the optional radar sensor to measure the distance of a car in front.

The additional feature has an impact on other use cases, which is modelled in the textual description of the affected use cases. An example for the use case “keep velocity” is given in the following subsection.

During application engineering, for each variant use case, it is decided whether the use case is (or is not) supported by the product to be built. The instantiation is done then with the help of the decision model. Further information on decision modelling can be found in [Atk01]. If a cruise control without distance regulator is built, all the variant use cases are removed, and the resulting use case diagram is a diagram without variability as shown above.

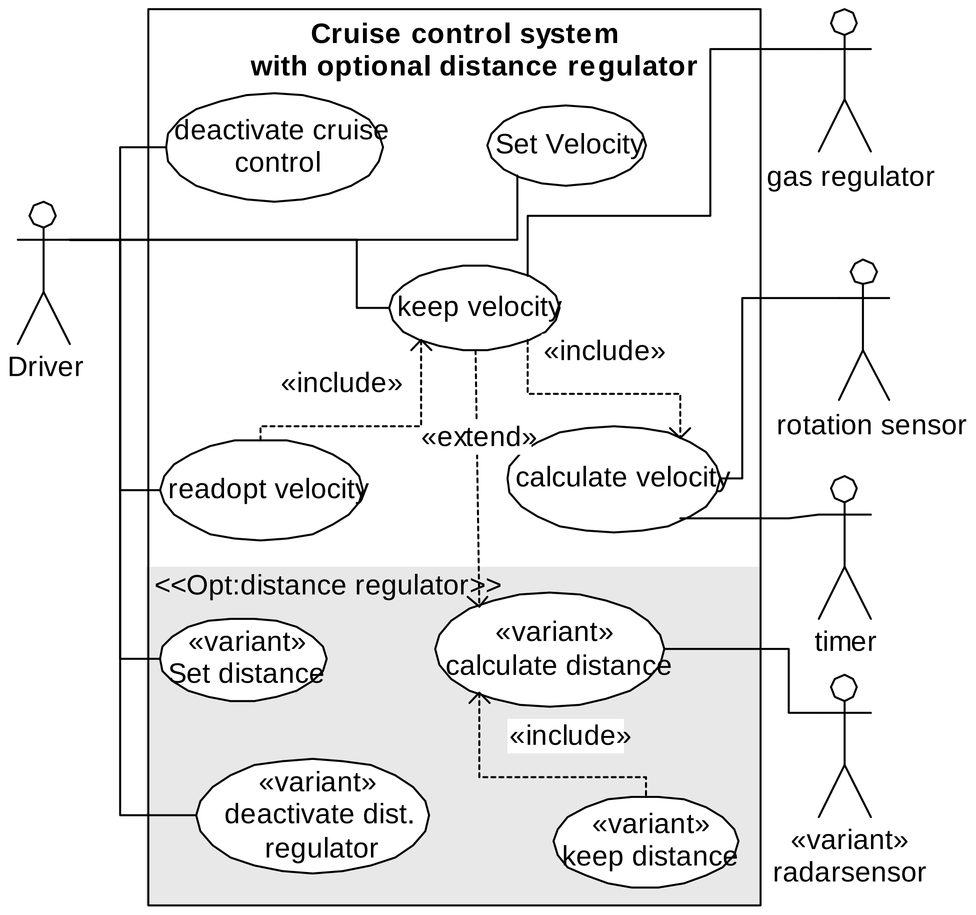


Figure 5-16 A variable Use Case Diagram

Generic Textual Use-Cases

In a textual use case description any text fragment may be variant. Variant text fragments are explicitly marked by pairs of the XML-like tags <variant> and </variant>. Figure 5-17 shows an example modelled with this approach, the use case "keep velocity".

The underlined questions in the use cases reflect the parts of the overall decision model that are relevant for this particular use case. This information is useful in both workproducts: integrated in the use case description, it helps to understand the use case's variability from the use-case's point-of-view, in the decision model, it helps to understand the variability of the whole product family and what the impact of a particular variability is (e.g., it has an impact on this use case). Hence the overall decision model captures the relationship between the decisions related to the above generic use case diagram and the textual description in Figure 5-17. That is, if the feature "distance control" is excluded the four variant use cases are removed and all variant text fragments are removed from the description of the use case "keep velocity", as well as the first alternative for step 4 is selected. This instantiation leads to the use case description given in the previous section.

Use Case Name: keep velocity
Short Description: keep the actual velocity value over gas regulator
<variant> by controlling the distance to cars in front </variant>
Actors: driver, gas regulator
Trigger: actor driver, <variant> actor distance regulator </variant>
Precondition: --
Input: starting signal, velocity value vtarget
Output: infinit
Postcondition: vactual = vtarget
Success guarantee: vactual = vtarget
Minimal guarantee: The car keeps driving
Main Success Scenario:

```
    1.) <keep velocity> is selected by actor driver
    2.)get vactual, vtarget (<Calculate Velocity>)
    3.) Does a distance regulator exist?
    <variant OPT> get dactual, dtarget  (<Calculate Distance>) </variant>
    4.) Does a distance regulator exist?
    <variant ALT 1: no; only cruise control>
            - compare vactual and vtarget
            If  vactual < vtarget : gas regulator increase velocity
            - restart <keep velocity>
            If  vactual < vtarget : gas regulator decrease velocity
            - restart <keep velocity>
            else restart <keep velocity>
    </variant>
    <variant ALT 2: yes, cruise control + distance regulator>
            - - compare vactual and vtarget
            If  vactual < vtarget : gas regulator increase velocity
            - restart <keep velocity>
            If  vactual < vtarget and atarget < aactual: gas regulator decrease
            velocity  (a : acceleration )
            - restart <keep velocity>
            If  vactual < vtarget and atarget > aactual: gas regulator increase
            velocity
            - restart <keep velocity>
            else restart<keep velocity>
    </variant>
```

Figure 5-17 A Variable Textual Use Case

5.2.3.4 Decision Modelling

Whether a use case in a use case diagram is an optional use case or whether it is an alternative to another use case is captured outside of the use-case diagram in a decision model. This is done simply because this information would overload the use-case diagram, make it less readable and thus less useful.

Table 5-10 shows an excerpt of the decision model in textual form for the use case diagram and the textual use case. As in this example the variation point is rather simple (Does a distance regulator exist or not?) the decision model also is very simple. If there is more and more complicated variability and hierarchical decisions, the decision model gets more complex and can really support the software engineers during application engineering.

Variation Point	Decision	Actions
1	The car has no distance regulator	Remove Use Case “set distance“ from use case diagram
		Remove Actor “radar sensor” from use case diagram......
		Remove variant <variant Alt 2 > from use case “keep velocity”
		Remove the tags <variant Alt 2> and <variant Opt > from use case “keep velocity” point 4......
	The car has a distance regulator	Remove the <<variant>> tag from all use cases in the use case diagram
		Remove the <variant Opt > tag and the </variant> tag from use case “keep velocity” point 3......
		Remove <variant Alt 1> from use case “keep velocity” point 4......

Table 5-10 A partial decision model

5.2.3.5 Results

In our experience, use cases are a good means to elicit, structure and represent user-level information during the requirements phase. Extended with variation points, they also allow people to easily switch from single-system requirements-engineering practices to domain analysis. The produced generic use-cases also support and guide application engineering (in particular, its requirements phase). Thereby, each variation point must be instantiated, that is, generic use-cases with variation points are systematically transformed into "normal" single-system use-cases as individual customers expect them. The approach makes explicit the functionality of the system from the end-user perspective, which might be complementary to conventional feature modelling focusing on end user features. In general, the described approach pushes the explicit consideration of variability to the early phases of product-line development, which is required to systematically manage and evolve a product-line infrastructure. With the right decision model that captures the relationships and dependencies among variation points, the approach captures the traceability paths from variant use-case actions down to variant implementation elements.

5.2.4 A model for eliciting product line requirements from existing documents

In this section, we describe a conceptual model or metamodel that can be used for the extraction of common and variable requirements for a product line. In general, a metamodel is “an information model for the information that can be expressed during modelling” [meta]. This conceptual elicitation model consists of models describing the syntactic and semantic types of information found in user documents and requirements specifications.

The elicitation model consists of four parts (see Figure 5-18):

- A user documentation model describing the elements that are typically found in user documentations, manuals and technical specifications (e.g., sections, glossaries, and lists).
- A requirements concept model describing concepts that are typically used in requirements specifications (e.g., roles, activities, functions) independent of the notation used.
- A variability concept model describing the principle commonality and variability concepts that can be found by comparing different documents and that are used for modelling.
- A requirements artefact model describing elements of typical single system requirements specifications and product line models. These elements form a notation that is used to capture requirements (like Use Case elements, features or textual requirements). Those requirements can have, but do not have to have an explicit representation of variability.

The transition from one stage of the model to another stage is described by heuristics (specific rules-of-thumb or arguments derived from experience), see [JoDo03a] and [JoDo03b]. It is also possible to directly transform requirements concepts into requirements artefacts without searching for variabilities (see arrow “condensed heuristics” in Figure 5-18).

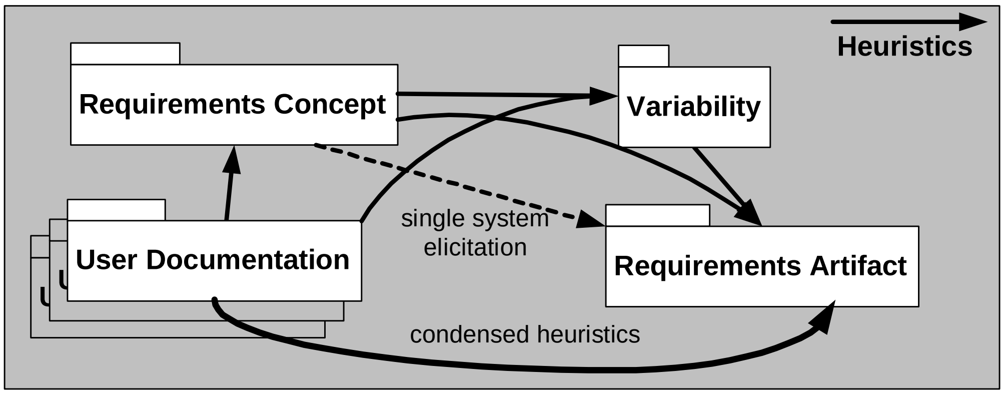


Figure 5-18 Overview of the model

5.2.4.1 User Documentation Model

Our user documentation model (see Figure 5-19) describes the principal constituents of user documents. The document types that we analyze are user documentations or user manuals that describe the functions and usage of a system and product descriptions that describe the features and technical details of a product. A document normally has a title, it often has a table of contents and a glossary and it consists of several sections. A TOC entry normally corresponds to a heading in a section. A glossary consists of a list of terms that are described in paragraphs. A paragraph consists of sentences; it can also contain figures, tables and formulas. A sentence is composed of phrases (language constructs consisting of a few words) and/or words. A phrase can also be a link (describing a reference to something inside or outside the document). Most elements of the user documentation model have attributes describing characteristics of this element (like highlighted for paragraphs and words, or numbered for lists), the attributes are not shown in the figure. This model describes the

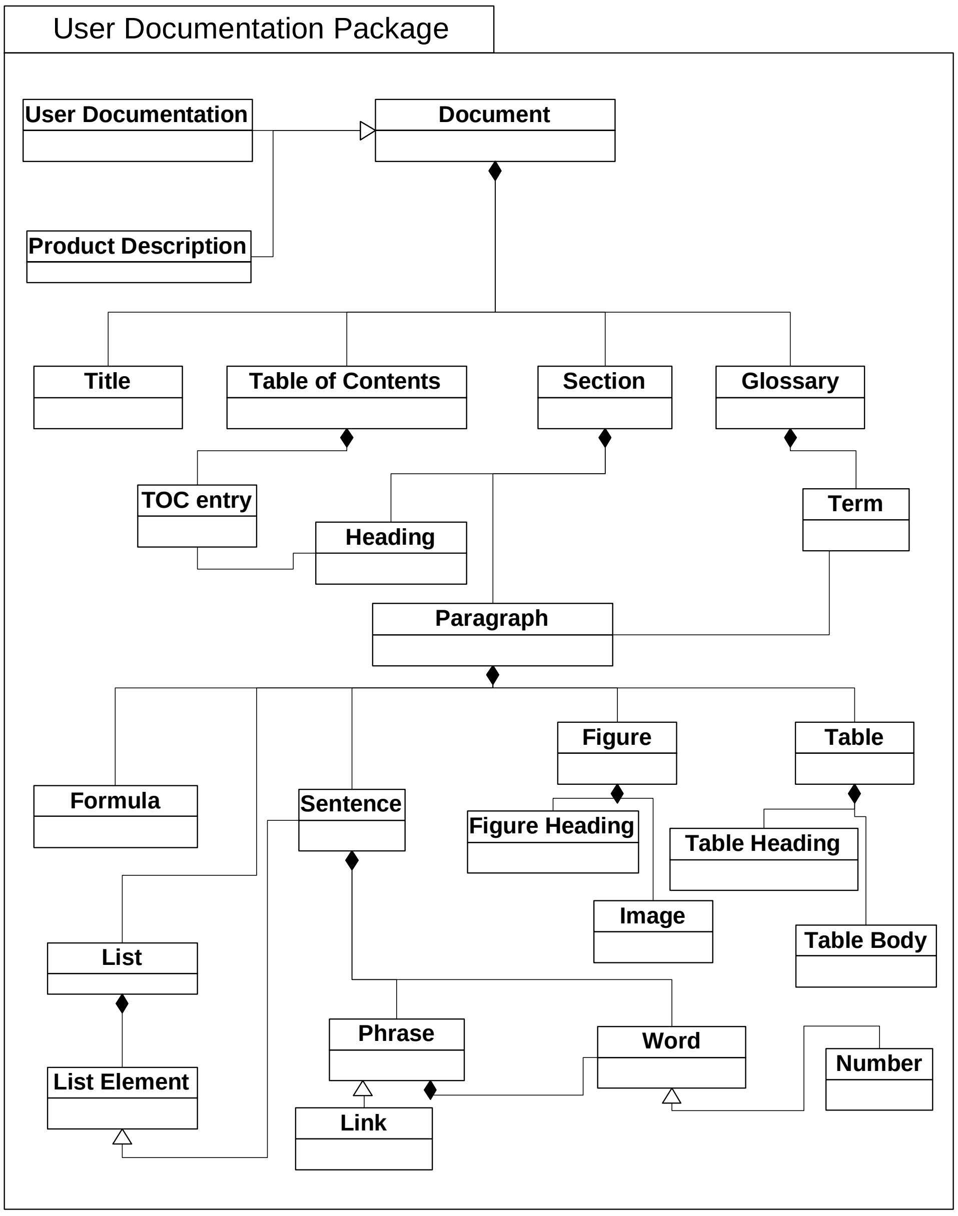


Figure 5-19 Model of User Documentation

elements of a document on an adequate level for eliciting requirements concepts.

The user documentation package consists of the following elements:

- **Document**
  According to [whatis] “a document (noun) is a record or the capturing of some event or thing so that the information will not be lost. Usually, a document is written, but a document can also be made with pictures and sound. A document usually adheres to some convention based on similar or previous documents or specified requirements…A document is a form of information.”. In our case a document contains structured information that is organized in sections and paragraphs and consists mainly of words (opposite to a picture, that can also be a document)
- **User Documentation**
  According to [Som01] there are different types of user documentation: a functional description, an installation document, an introductory manual a reference manual and an administrator’s manual. As we focus rather on the characteristics of the running system than on the installation, user documentation is for us at first hand the functional description, the introductory manual and the reference manual. Anyway, if installation of the system and the capabilities and features of installing the system should be investigated, the installation document or the administrator’s guide can also be in focus
- **Product Description**
  A product description is a short functional description of all the capabilities of a system. Such description can often be found under “featurelist” or “Technical description” on companies websites.
- Elements of a Document:
  - **Title**
    The title of a document usually is on the first page and describes the content of the document in a short and concise phrase. Not all documents have a title.
  - **Table of contents**
    The table of contents of a document contains a list of headings and often chapter and page numbers.
  - **Section**
    A document normally is spilt in sections that contain a heading and the actual text. Most documents (but not all) consist of sections.
  - **Glossary**
    Often technical documents have a glossary. In the glossary terms that are introduced in the document and that are specific for the product or the domain of the product are described. A glossary entry consists of a glossary term and a paragraph that describes the glossary term.
  - **Glossary Term**
    A glossary term is a term in the glossary
- Elements of a Section
  - **Heading**
    A heading describes the content of a section in a short concise phrase. Heading can be main headings (numbered 1, 2, 3 etc.) or subheadings (numbered 1.1; 3.2.1 e.t.c). Remark: We do make a distinction between headings and subheadings as they have the same meaning for the elicitation process.

- **Paragraph**
  A section consists of a heading and several paragraphs. Normally a paragraph consists of 1 to 20 lines (some paragraphs can even be longer) and is ended with a carriage return, a blank line or a paragraph break.
- **Formula**
  A formula is a special part of a of paragraph that contains numbers and mathematical symbols.
- **List**
  A list is a special part of paragraph (or a whole paragraph) that contains a numeration that is marked with numbers, dots or hyphens.
- **Sentence**
  A sentence is a grammatical unit that is composed of one or more clauses that include at minimum, a predicate and an explicit or implied subject, and expresses a proposition.([c.f. [Loo99]).
- **Figure**
  A figure consists of an image displayed in the document and figure heading (optional) describing the content of the image.
- **Table**
  A table consists of a tabular structure containing words, sentences or phrases and a table heading (optional) describing the content of the table.

- Elements of a Sentence
  - **Phrase**
    A phrase is a syntactic structure that consists of more than one word but lacks the subject-predicate organization of a sentence or clause. ([c.f. [Loo99]).
  - **Word**
    A word is a unit which is a constituent at the phrase level and above. It is sometimes identifiable according to such criteria as being the minimal possible unit in a reply.([c.f. [Loo99]).
  - **Number**
    A number is a counting word.

With the help of these document types and document parts, each kind and part of user documentation can be classified according to this conceptual model. With the help of this classification and the other models described in the subsequent sections the elements can be generalized and transferred to parts of product line models.

5.2.4.2 Requirements Concept Model

The requirements concept model describes concepts that can be elicited from user documentation and that are normally realized or described by requirements artefacts in requirements specifications. The model describes the elements independent of a specific notation (like textual or Use Case representation). The most general requirements concept is a requirements element. A requirements element can be everything that is of value for a requirements specification. A requirements element can be a user task, a role, data, a naming convention, a constraint or a relation to something in the environment of the system to be described. Data can either be I/O data or internal data. Constraints can either be usage or design constraints. A user task, that describes the high level task the user wants to perform with the help of the system can be decomposed into activities, activities consist of navigation elements, system functions and a mapping of the activities to functions.

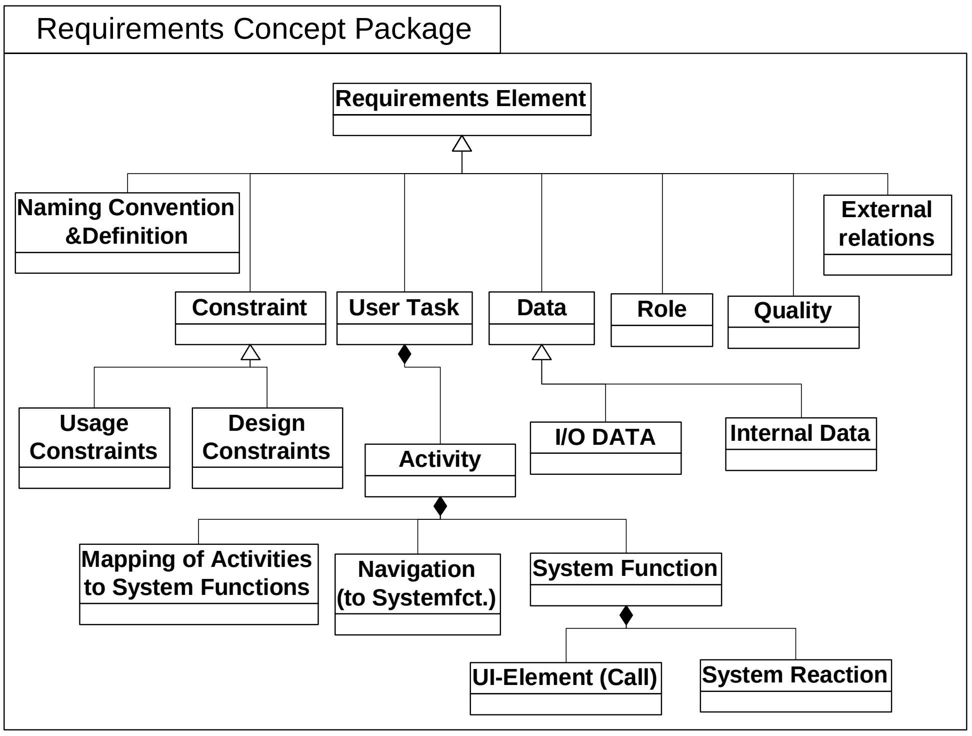


Figure 5-20 Requirements Concept model

In the following, each requirements element is described in more detail:

- **Naming Conventions and Definitions**
  The definition of a term or name for an object in the product domain (e.g., a term defined in a glossary).
- **External relations**
  The relation of an object of the product or the product itself to an object in the environment (e.g., the mobile phone to a headset via an interface).
- **Constraints**
  A given constraint on the system (either by physics, laws or also marketing).
- **Usage Constraints**
  Constraints on the usage of the system (e.g., the user is not allowed to operate a mobile phone in the plane).
- **Design Constraints**
  Constraints on the design of the system (e.g., there have to be several subsystems that can be installed stand-alone).
- **Data**
  Any kind of data that is produced or consumed by the system.
- **I/O DATA**
  Data that the system sends to or gets from the environment. (e.g., the maximum download stream of a mobile phone from a network node).
- **Internal Data**
  Data that is only used inside the system, i.e., the data is not directly visible to the outside

(e.g., the data exchanged between a messaging and the network component inside the phone).

- **Role**
  The role a stakeholder of a system to accomplish a certain task (e.g., a person calling another person is in the caller-role).
- **Quality**
  A quality characteristic of the product, a process or a resource (e.g., the quality of the speech during a phone call).
- **User Task**
  User tasks are tasks, a certain user has to perform. They are supported by the system (e.g., "placing a phone call"), but include some user involvement. A task specifies what the computer and user shall accomplish together without indicating which actor performs which part of the tasks [Laue03].
- **Activity**
  An activity is more fine grained than a user task, carried out by the user (not necessarily with the system (e.g., dialling a number). Several activities are carried out to achieve a certain user task.
- **System Function**
  A function, the system provides to support a user in achieving a task (e.g., establishing a network connection or deliver an SMS to the network).
- **UI-Element (Call)**
  A user interface element is displayed by the system to be used by the user to activate a system function (e.g., "send message" in a menu to finally transmit the message).
- **System Reaction**
  The reaction of the system to a call of a system function (e.g., system shows the message "message sent").
- **Mapping of Activities to System Functions**
  The logical mapping of an activity that is carried out by the user to system functions, the user calls to help him achieve a certain task (e.g., activity "dialling a number" to system function "establish network connection").
- **Navigation (to System Functions)**
  A user interface element is displayed by the system to navigate to a system function (e.g., "properties" in the menu).

Based on this requirements concept model and the model of user documentation described in section 5.2.4.1 we can define heuristics for the transition of elements from one model to another. Example heuristics for transitioning from a user documentation element to a requirements element are: "A heading that contains a verb often is an activity" or "a highlighted sentence containing the phrase "normally" or "with the exception" can describe constraints".

5.2.4.3 Variability Model

In the variability model, the variation aspects are described. In order to find different variability elements, the requirements elements (from the requirements concept model) found in different user documentations are compared. The variability model is a product line specific model as it describes commonality and variability between different products. Variabilities can normally be found by comparing different documents. We decided to support the following variability elements and kinds of variation:

- **Commonality**

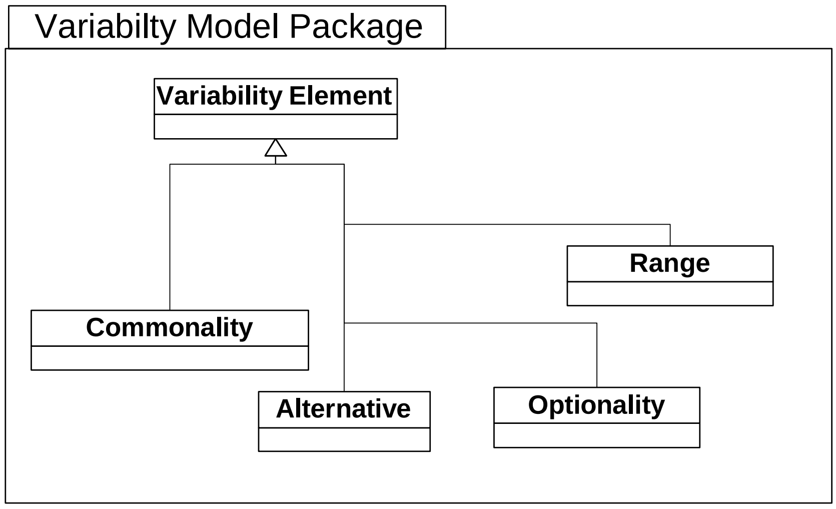


Figure 5-21 Variability Model

No variation exists in the requirements element, the same requirements element can be found in all documentations (i.e., it is present in every product).

- **Optionality**
  A requirements element exists in one or more products, but does not exist in all products.
- **Alternative**
  An alternative is expressed over a set of at least two requirement elements. It expresses that exactly one requirement element of the set is present in a product of the family. (e.g. one product supports an "Oracle" database, one product supports a "Pointbase" database).
- **Range**
  There is a range of values for a requirements element which is supported by the different products (e.g., the memory size can vary from 10 to 128 MB). Each product instantiates the requirements element with a certain value of that range (e.g., the memory size for product A is 128 MB).

Based on those variability elements, heuristics can be defined that identify different variable requirements concepts by comparing the user documentations of several legacy products. These heuristics are depicted by the two arrows in Figure 5-18 from user documentation and from requirements concept to variability.

Examples of such heuristics are "numbers in the document that were identified as data and belong to the same function and that have a different value can be a range variability element" or "navigation elements that occur only in one documentation can be a hint for an optionality (an optional user interface element)".

5.2.4.4 Product Line Artefact Model

The fourth package of our conceptual elicitation model is the requirements artefact model (see Figure 5-22). In this model, different elements of requirements specifications that can be used for single system modelling and for product line modelling, are described. Different from the requirements concept model, that describes the elements on a conceptual or semantic level, the requirements artefact model describes requirements elements on a syntactic or notational level. In different kinds of requirements specifications, the same conceptual elements can be described with different notational elements, e.g. a role from the requirements concept model can be an actor in a Use Case description or a stakeholder description in a textual requirements specification.

As we also describe the application of our approach for product line modelling, we have an integrated model of variability here. The variability model we use here is the model described in the thesis of Muthig [Mut02]. The model described by Dirk Muthig is a model to describe generic product line assets. Product Line Assets can be product line artefacts, describing the product line itself and decision models describing the constraints on the product line artefacts. Figure 5-23 shows the general structure of the metamodel for product line information. The grey boxes indicate the two subpackages that were defined to improve the structure of this model. The first package, *ProductLineArtifact* contains the definition of product line artefacts.

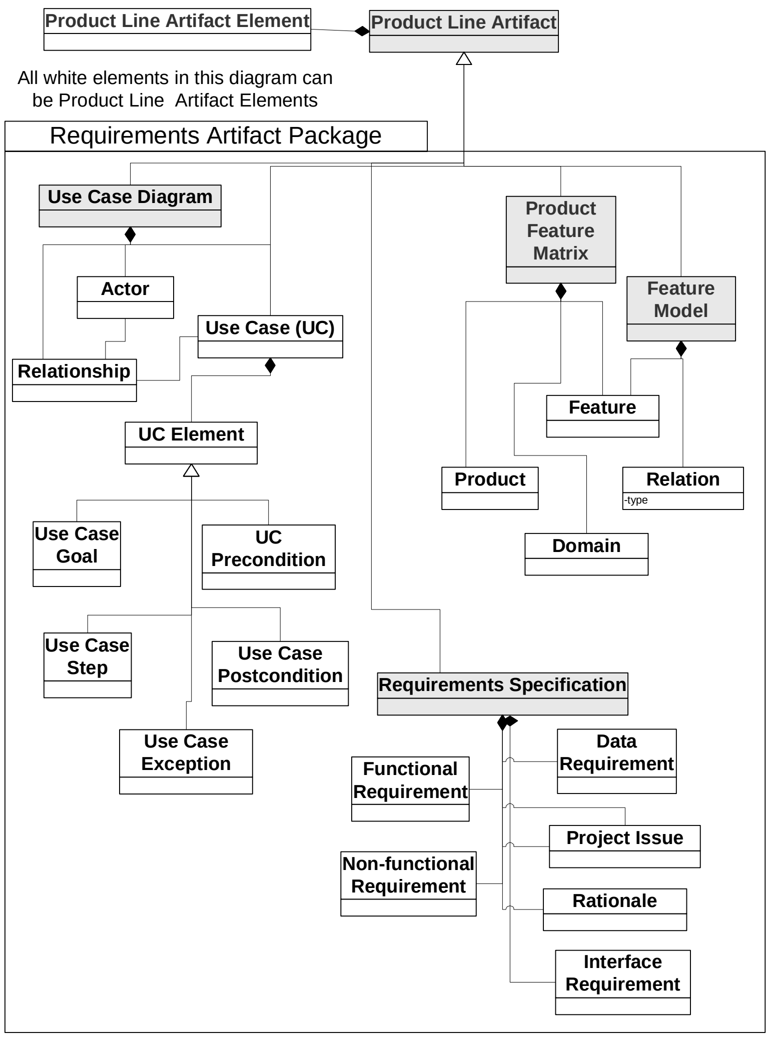


Figure 5-22 Requirements Artefact Model

The second package *DecisionModel* depends on the first package and defines the concepts of a decision model with respect to the definition of product line artefacts.

In product line engineering, variability has to be made explicit in the requirements artefacts. Different extension (e.g. to UML-Use Case diagrams or to textual Use Cases) exist that make the variability explicit and give support for instantiation of requirements for application engineering. Some of these extensions use stereotypes or tags to describe variability, some extensions use extra elements to make variability explicit.

As variability is encapsulated outside the requirements artefact model in the product line artefact and the product line artefact element (see Figure 5-23), the model can also be used for specifying single systems requirements. At the moment we have specified different kinds of requirements notations: Use Cases, textual requirements specifications and the product line specific notations product feature matrix [Schm02] and feature model [Kan90]. Further requirements artefacts will be integrated into the requirements artefact model. We added different representations here, as our general approach to product line modelling is customisable and highly depends on the requirements elements found in the organization that wants to do product line engineering. For doing product line engineering, we put variability elements on top of the existing notation and so can keep the notation similar to the one used in the organization before. According to the elements in Figure 5-17, a Use Case diagram consists of Use Cases, actors and different relationships between the Use Cases and the actors. A textual Use Case (according to Cockburn [Coc01]) consists of different elements like Use Case goal, precondition/ post condition, Use Case exceptions and the actual description of the Use Case consisting of steps. The form of requirements specification we describe here follows the IEEE Standard 830. A requirements specification is a textual document consisting of functional, non-functional and data requirements including project issues and rationales for the different requirements.

The requirements artefact model described in Figure 5-22 consists of the following elements:

- **Product Line Artefact Element**
  A product line artefact element can be every documentation element that can be described to document a product line (in our case to document requirements on a product line or a product line requirements model).
- **Product Line Artefact**
  A product line artefact consists of product line artefact elements. It is an artefact that captures product line concepts such as commonalities or variabilities in an integrated and explicit form. A product line artefact that captures no variabilities is identical to an artefact used in a single-system context [Mut02].
- **Use Case Diagram**
  A use case diagram shows the relationship among actors and use cases within a system. A use case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalizations among the use cases.
- **Actor**
  An actor is a person or another entity that interacts with the system.
- **Relationship**
  Relationship between actors and use cases or between use cases describe that there is interaction between the elements.

- **Use Case**
  A use case is a collection of possible sequences of interactions between the system under discussion and its Users (or Actors), relating to a particular goal. The collection of Use Cases should define all system behaviour relevant to the actors to assure them that their goals will be carried out properly. Any system behaviour that is irrelevant to the actors should not be included in the use cases [Coc01]. A use case is either a bubble in a use case diagram (then it is not further specified) or described as a textual use case, then it consists of further use case elements. How Use Cases can be extended with variability (so, how to inherit product line artefact element properties into the use case) is described in Section 5.2.3.3
- **Use Case Element**
  A textual Use case has certain elements: Use case goal, precondition, postcondition, use case steps and extensions (these elements follow the division by Cockburn [Coc01]
  - **Use Case Goal**
    The Use Case Goal describes in a few words what the use case does.
  - **Use Case Precondition**
    The Use Case Precondition describes the situation at the start of the use case and which conditions have to hold so that the Use Case can be started.
  - **Use Case Postcondition**
    The Use Case Postcondition describes the situation at the end of the use case and the minimal success guarantee.
  - **Use Case Step**
    A Use case consists of a sequential order of steps. Often this is a sequence of

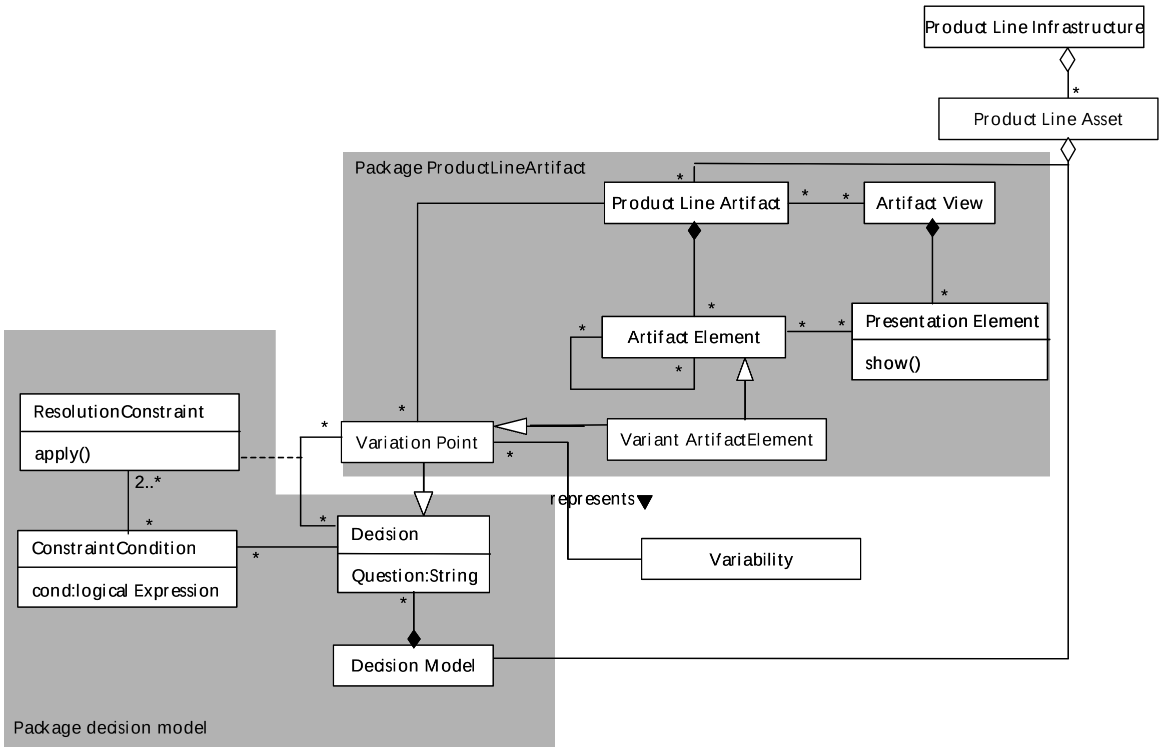


Figure 5-23 General structure of the meta model for product line information (from [Mut02])

actions of the user and system reactions.

- **Use Case Exception**
  In Use Case Exception all exceptions or special situations that can happen during the steps of the use case are described.

- **Requirements Specification**
  Another way to capture requirements on a system is a textual requirements specification. A textual requirements specification can be organized with different schemata, e.g. [IEEE98] or [Robe99]. Our conceptual model contains elements that can typically found in requirements specifications and that can be found in user documentation. Requirements are typically numbered. They shall contain a date, a priority, a change history e.t.c.
  - **Functional Requirement**
    Functional requirements describe the functionality of the system in a structured way**.**
  - **Non Functional Requirement**
    A non functional requirement describes a characteristic that the system shall have.
  - **Data Requirement**
    A data requirement describes a requirement on the kind or structure of internal data.
  - **Interface Requirement**
    An Interface Requirement describes how the interfaces of the system shall be realized and how the interfaces of other systems to the system to be developed look like.
  - **Project Issue**
    Project issues are all things that are related to external constraints, like laws or that are related to performing the project of developing the system like personnel constraints.
  - **Rationale**
    Rationale describes why a certain requirement was chosen.
- **Product Feature Matrix**
  With the help of features and the envisioned products and subdomains of the planned system, a distribution of features into products, a so called product feature matrix [Schm02] can be made.
- **Feature Model**
  A Feature Model [Kan90] describes the common and variable capabilities of the systems in a product line and their relations in a graphical form. It captures the decisions that have to be made when instantiation a product out of the common infrastructure of the product line.
- **Feature**
  A feature describes a capability of the system that is of importance for the user. A feature is part of the product feature matrix and of a feature model.
- **Relation**
  Features in a feature model can have different relations (e.g. one feature requires a different feature, two features are mutual exclusive e.t.c)
- **Product**
  The planned products of the product line are part of the product feature matrix
- **Domain**

A product line consists of several subdomains, some of them are visible for the user, some of them are internal domains. A domain can be later e.g. realized as a component.

5.2.4.5 Using the conceptual model

We have defined heuristics for transitioning requirements concepts into requirements artefacts (e.g., "a role is described as actor in a Use Case diagram") and heuristics that additionally include variability (c.f. Figure 5-18). An example of such a heuristic also considering variability is "an optional activity can be represented as an optional Use Case in a use diagram". For the transition between elements of these packages we have found different heuristics. For users of the approach and the conceptual model those heuristics can be integrated to condensed heuristics describing the transition from user documentation directly to requirements artefacts (c.f. arrow "condensed heuristics" from user documentation to requirements artefact in Figure 5-18). Between all four parts of the model, heuristics can be defined to describe how elements are typically converted from one part of the model to another. A Heading from the user documentation model can be a user task in the requirements concept model and can then be physically represented as a feature. The transition rules for transitions between the different models will be further elaborated and published in the future.

Additionally, a tool is planned that will realize the conceptual models. The tool will get user documentation as input and give suggestions for product line artefacts to the domain experts based on this user documentation. This tool will be described in detail in the work package 5 deliverables of the Empress Project.

5.2.5 Conclusion

In this section we described the general concepts of product line engineering and especially requirements engineering for product lines. We described an approach here to extend use cases to be able to model common and variable requirements on a product line and described a conceptual model that supports requirements elicitation for product lines based on user documentation. The conceptual model consists of four parts : A user documentation model describing the elements that are typically found in user documentations, a requirements concept model describing concepts that are typically used in requirements specifications, a variability concept model describing the principle commonality and variability concepts that can be found by comparing different documents and that are used for modelling and a requirements artefact model describing elements of typical single system requirements specifications and product line models.

The conceptual model forms the basis for describing requirements on a product line with use cases with commonalities and variabilities as described in section 5.2.3. With the help of transformations based on the conceptual model, elements from user documentation can be integrated into product line models, describing requirements on a product line.

The conceptual model will serve as a basis for a tool that semi automatically elicits product line model elements from user documentation. This tool will be part of the demonstrator developed in WP 5 of the Empress Project. Within WP5, the applicability of the elicitation and modelling approach described here will be shown on an example of different user manuals from Daimler Chrysler "Instrument Cluster" (see Deliverables WP5.1).

5.3 A Method for Eliciting, Documenting, and Analyzing NFRs

5.3.1 Introduction

Requirements engineering approaches have for a long time mainly focused on functional requirements. During the last 5 years, several approaches dealing specifically with non-functional requirements have emerged. They support the elicitation, documentation, verification and validation of non-functional requirements: sometimes only concentrating on the non-functional requirements, sometimes in conjunction with functional requirements, and sometimes in conjunction with architecture. The position we put forward in this deliverable is that functional requirements, non-functional requirements (NFRs), and architecture options must be treated together. Furthermore, non-functional requirements have to be refined to a level that they can be measured. Otherwise it is impossible to validate the fulfilment of the non-functional requirements or, as an intermediate step, if a specification of a system is on track to fulfil the requirements. The validation of the fulfilment of each non-functional requirement shall be as early as possible. Therefore, Fraunhofer IESE and Siemens developed the Empress method for eliciting, documenting and analysing non-functional requirements. This method proposes a refinement process and documents supporting this process (e.g. checklists) that strongly takes into account the dependencies between functional, non-functional, and architectural decisions. The main goal of our method is to achieve a minimal, complete, and focused set of measurable and traceable NFRs as early as possible.

Our approach provides:

- A quality model that captures general characteristics of quality attributes, metrics to measure these quality attributes, and means to achieve them. In particular, this model reflects views of different stakeholder roles, such as customer and developer. This quality model supports measurability, completeness as well as focus due to the views.
- A distinction of different types of quality attributes, which gives guidance on how to elicit NFRs. This specific treatment for the various types supports focus of the NFRs.
- Detailed elicitation guidance in terms of checklists and a priorisation questionnaire. The former are derived from the quality model and the types of quality attributes and help to elicit efficiency NFRs in concert with use cases and a high-level architecture. The latter is used to prioritise high-level quality attributes (i.e., maintainability, efficiency, reliability, usability). The checklists support completeness, the priorisation questionnaire supports the focus of the NFRs.
- Documentation guidance by providing a document structure and templates.
- The use of rationales to justify each NFR. Using rationales supports minimality of the set of NFRs.

Figure 1 gives a simplified view on our approach.

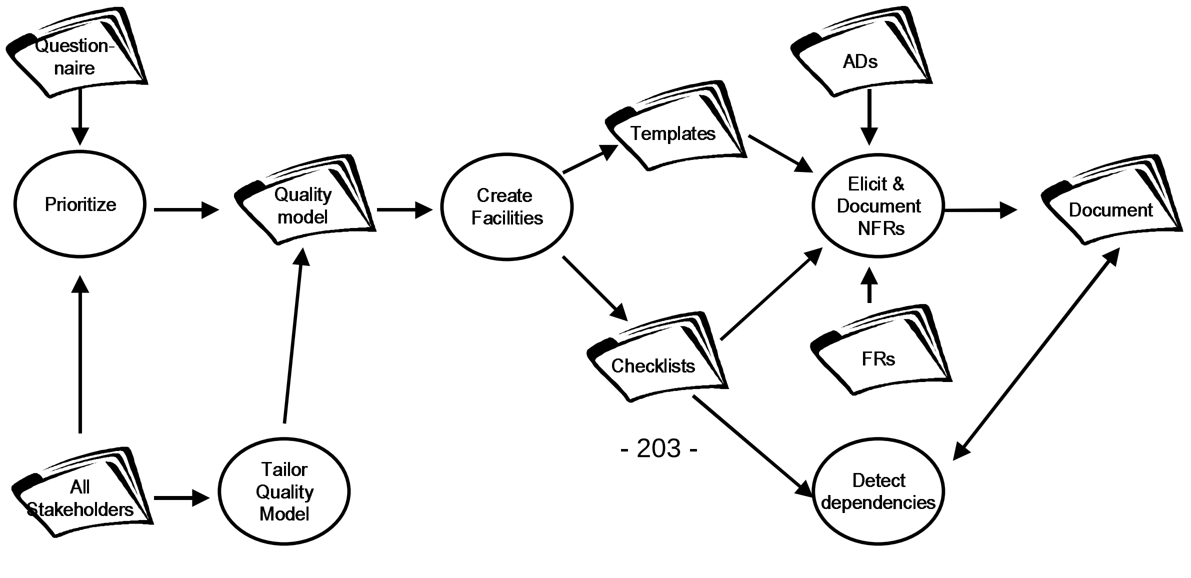


Figure 5-24: Simplified view on process

This chapter is structured as follows. In Section 2.2, we sketch our terminology and lay the foundation of our approach by explaining a metamodel for quality modelling. Furthermore, we explain how to develop the quality models, and checklists for the NFR elicitation. In Section 2.3, we describe how to prioritise the high-level quality attributes and how to elicit, document and consolidate the NFRs. In Section 2.4 experience based quality models are described that can be used in our approach for eliciting the NFRs. As these quality models are used for making NFRs measurable, they can also be used to analyse a system with regard to the high level quality attributes. In Section 2.5 we conclude and describe our future work.

5.3.2 Foundation and Terminology

In this section, we present the foundation of our approach. First, we present the document structure we assume for the requirements engineering phase. In Section 5.3.2.2 we define important concepts of our approach like quality attribute, NFR, and means and present the according meta-model. Section 5.3.2.3 and 5.3.2.4 show how to develop and adapt quality models, checklists used for elicitation and documentation, and the templates.

5.3.2.1 Underlying document structure

As basis for our approach, we use a document structure that was also used in the Quasar project (see [DS] for details). This document structure separates requirements into four different levels of abstractions:

- The system requirements document describes the viewpoint of the contractor.
- The system specification document captures the detailed functionality of the whole system (including software and hardware), but abstracts from the details of the real sensors, actuators and user interfaces.
- The software requirements document captures the requirements on the software as part of the overall system.
- The software specification document describes the detailed functionality of the software necessary to interact with specific sensors and actuators determined in the hardware design.

In the following, the elements of requirements documents and specification documents are described in more detail. For this description, we abstract from system and software:

- **Requirements document**: The requirements document states functional and non-functional requirements as well as architectural options and first architectural decisions with their rationales and relationships between and within these types of elements. The requirements and architectural decisions are on a level that enables the contractor to determine if the actual system fulfils these requirements. Furthermore, this requirements document enables the contractor to validate that the specification document meets their requirements.
- **Specification document**: The specification document describes detailed functional and non-functional requirements that state how the requirements of the requirements document are realized. Furthermore, options, as well as architectural decisions, rationales for them and relationships between non-functional, functional and architectural decisions are described. The requirements and architectural decisions stated in the contractors specification document are on a level that enables the contractor to see if they meet the requirements stated in the requirements document (e.g., with the help of traceability, see [W3.2]). Furthermore, this specification document enables the contractor to validate that the actual system meets their requirements (written down in the requirements document).

In a subcontracting situation, which is typical in the domain of embedded systems, the contractor creates the requirements documents with focus on the problem definition. The subcontractor creates the specification documents with further problem refinement and solution refinements. This holds for system documents as well as for software documents.

For the description of the Empress method for eliciting, documenting and analysing non-functional requirements, we abstract from system and software (as they run analogous), we only distinguish between requirements documents and specification documents. This means that there are two instances of the Empress method for eliciting, documenting and analysing non-functional requirements, one used for creating the system requirements and specification and one used for creating the software requirements and specification (and of course, for creating the corresponding documents). Both instances are carried out in an analogous manner concerning the elicitation and documentation, but with some different focus (problem vs. solution refinement).

5.3.2.2 Terminology & Underlying meta-model

The metamodel (see Figure 5-25) describes the main concepts of the approach. Our experience showed that certain decisions have to be made during the elicitation of NFRs (e.g. does a quality aspect affect a user task, or rather architectural options?). The concepts described in the metamodel support these decisions. In the following, we explain the most important elements.

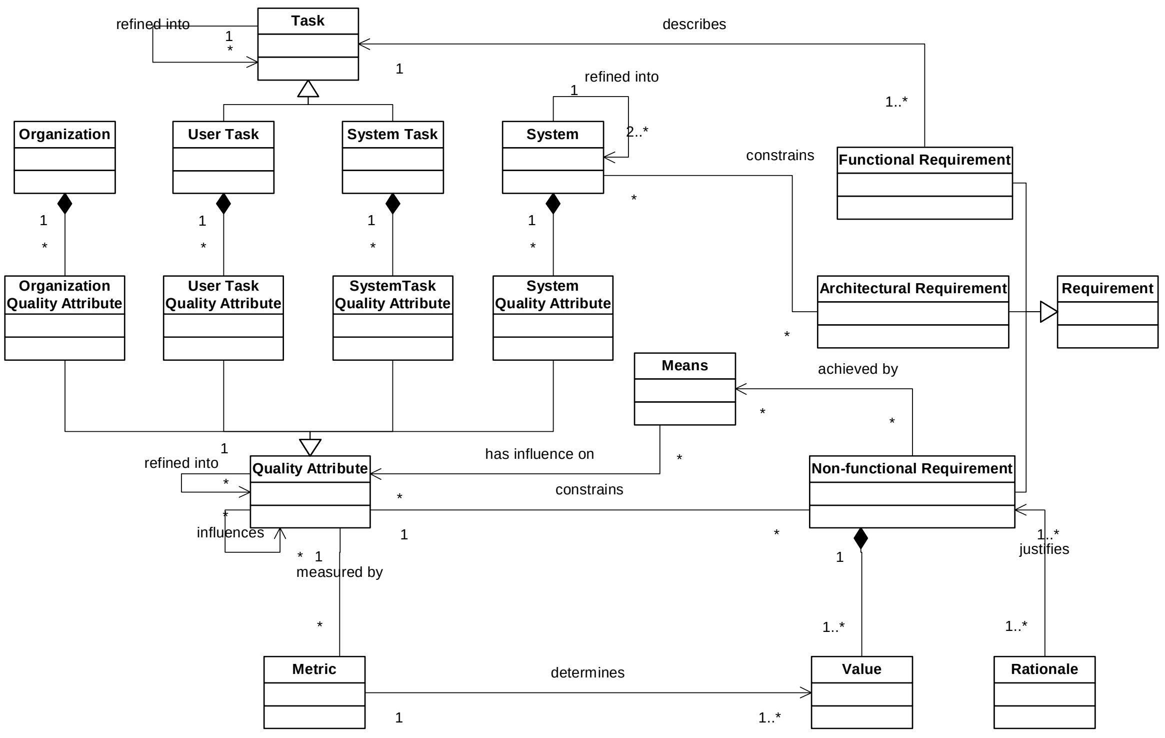


Figure 5-25 The Metamodel

- A quality attribute (QA) is a non-functional characteristic of a system, user task, system task, or organization. Quality attributes of the organization include development process specific aspects.
- The distinction between different types of quality attributes is important for our elicitation process. Each type of quality attribute is elicited differently (see section 5.3.3). QAs can be refined into further QAs. In addition, they can have positive or negative influences on each other. A more detailed description of the types of QAs and their relationships can be found in Section 5.3.2.3.
- A system (e.g., “wireless control and monitor system”) can be refined into a set of subsystems (e.g., “wireless network”, “mobile device”). Architectural requirements (e.g., “the system shall have a database”) constrain the system.

- We distinguish between two types of tasks: user tasks and system tasks. User tasks are tasks, a certain user has to perform. They are supported by the system (e.g., “monitoring of certain machines”), but include some user involvement. System tasks are tasks the system performs. In contrast to user tasks, the user is not involved in system tasks. Tasks can be refined into further tasks. Furthermore, user tasks can be refined into parts carried out by the user and system tasks (e.g., a user task “monitoring machine x” is refined into a set of system tasks such as “system displays alarm message if machine runs out of filling”). A task is described by one or more functional requirements (FRs).
- A NFR describes a certain value (or value domain) for a QA that should be achieved in a specific project. The NFR constraints a QA by determining a value for a metric associated with the QA. For example, the NFR “The database of our new system shall handle 1000 queries per second.” constraints the QA “workload of database”. The value is determined based on an associated metric “Number of jobs per time unit”. For each NFR, a Rationale states reasons for its existence (e.g., “the user will be unsatisfied if it takes more than 2 seconds to display alarm message”).
- We distinguish problem-oriented refinement from solution-oriented refinement. Problem oriented refinement is a refinement of NFRs on a higher level QA to NFRs on lower level QAs. E.g., a non-functional requirement expressed over the quality attribute “Time Behaviour” will typically be a requirement stating a problem (e.g. “The system shall have a good time behaviour.”). One can now refine this problem requirement via non-functional (problem) requirements expressed over the quality attributes “Boot Time” (e.g. “The system shall boot up in 2 min.“ and “Response Time” (e.g., “The system shall respond to an input A within 5 sec.” A solution-oriented refinement is made explicit in terms of means. A means is used to achieve a certain set of NFRs. In many cases, a means describes an architectural option that can be applied to the architecture to achieve a certain QA (e.g., “load balancing” is used to achieve a set of NFRs concerning the QA “workload distribution”). However, a means can also be process related (e.g., the means “automatic test case generation” is used to fulfil NFRs regarding “reliability”). Means are most times selected and used by the developers (e.g., Use of Design Patterns), but can also be visible for and demanded by the customer (e.g., Documentation).

5.3.2.3 Developing and adapting quality models

A quality model instantiates parts of our metamodel. It describes typical refinements of high-level QAs into more fine-grained QAs, metrics, and means. The idea of the quality model is to refine QAs to a measurable level, i.e., to QAs a metric can be attached. In addition, it describes relationships between different QAs. Therefore, it captures experience of previous projects. Our quality models are similar to the goal graphs for NFRs by [Chu00], but emphasize dependencies, and distinguish between different types of QAs. Quality attributes as input for these quality models were taken from industrial experience, but also from literature and standards as [ISO01]. Figure 5-25 for example, is an excerpt from a quality model for "efficiency".

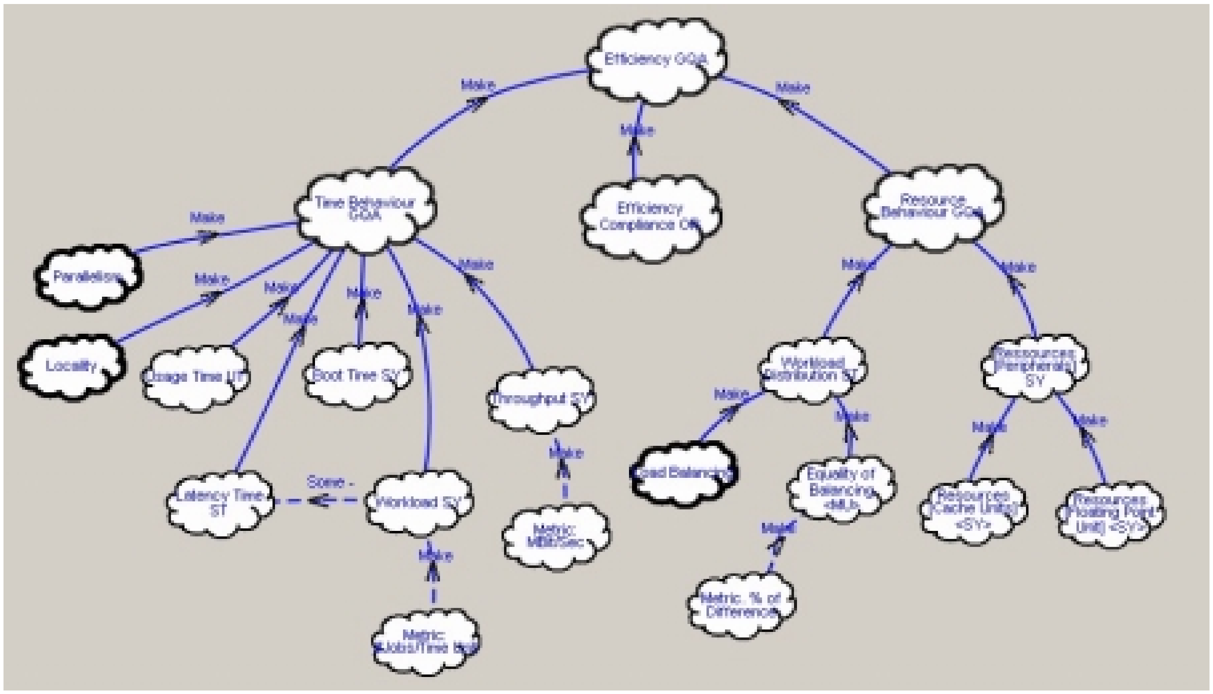


Figure 5-26 Quality model (e.g. for efficiency)

In Figure 5-26, QAs are represented by clouds with thin borders and the name of the QA plus a label to identify its type and without the word metric in it. Clouds with thick borders are means (called operationalization in [Chu00]) that have influence on the related QA and clouds with thin borders and the word metric in it are metrics associated to the quality attributes. There are basically five different types of QAs in this quality model (see also metamodel):

- General *QAs (GQA)* such as "Time Behaviour" are used to structure the QAs on lower levels.
- Organizational QAs (OR), such as "Experience", concern the organizational aspects. This also includes development process related aspects, such as required documentations, reviews, etc.
- System QAs (SY), such as "Capacity", are QAs related to the system and its subsystems (e.g., related to the database, secondary storage or network).
- User Task QAs (UT) such as "Usage Time", are related to tasks in which the system and the user are involved.
- System Task QAs (ST), such as "Response Time", are related to system tasks, i.e., tasks that are carried out by the system, not including the user any more (e.g., calculation of results).

Only the latter four QAs are constraint by NFRs. The first type of QA serves as a structuring for the hierarchical decomposition of the more fine-grained QAs. This structure is also used for the template for documenting the NFRs (see Section 5.3.2.4). How the NFRs for the QAs are elicited, depends on the type of the QA they constrain. This is described in Section 5.3.3. Furthermore, there is a distinction of QAs that are usually expressed by the customer and ones that are usually introduced by the developers (usage of angle brackets (<>) in the QA type label, like in Figure 5-26). E.g., the response time of a system is visible for a customer and the resource (capacity) of the cache units is typically only visible to the developer. This distinction is important to focus during the creation of the requirements document on customer visible QAs and during the creation of the specification document on how to refine the customer NFRs with the developer visible QAs and find solutions with the help of the means attached to the QAs. In Figure 5-26, some QAs are tagged with "MU", which expresses a Means Usage. Sometimes it is not possible to make a further refinement of the QAs without assuming that a certain means was used. In Figure 5-26, the QA "Equality of Balancing" is a refinement of "Workload Distribution", only if a "Load Balancing" Means is applied. In this case, the lower level QA is tagged with "MU" showing that it depends on the usage of a means. Nevertheless, the Quality Attribute can also be related to one of the categories organizational, system, user task or system task QA. In this case it is system oriented, as the load is distributed on two or more elements of the system architecture.

Four types of relationships can be found in such a quality model that relate the various kinds of QAs, means and metrics. The metamodel in Figure 5-25 describes the general types of relationships.

- Refine: A QA, such as "efficiency", is refined into more detailed QAs, such as "time behaviour" and "resource utilization".
- Influence (Means): A means has influence on a QA, i.e., it is used to achieve the NFRs constraining the QA. "Load balancing", for example, is influencing "workload distribution" and used to achieve the constraining NFRs (e.g., "The workload for computing the results must be equally distributed on the two processors").
- Measured by: A QA is measured by a metric. The "workload" can, for example, be measured by the metric "number of jobs per time unit".
- Influence (QA): A QA can be positively or negatively influenced by another QA. If the "workload", for instance, is higher, the "latency time " will increase (negative influence).

Origin of quality models
There are three possibilities how such a quality model appears:

- The company already has such a quality model, they want to use it as is. In this case, it is very important to understand what the quality attributes in this company specific quality model mean to the company.
- The company wants to use the default quality model provided with this approach (see section 5.3.6 for the models) without adaptations. Reasons for this can be a lack of time or money.
- The company wants to build a quality model that is tailored to their context. This should be the preferred version. Our recommendation is to build a quality model together with the company in a workshop. By doing so, the quality model benefits from the already integrated experience of our reference quality model and it is tailored to the project and company. Furthermore, it is important to include all related stakeholders in such a workshop. For example, when tailoring the quality model for maintainability, it makes sense to include the maintainer of the future system.

When creating and tailoring the quality models, several times, the question arises, whether classifications should be included in the quality models or not. For example, one could have the QA “Structuredness” refined into “Structuredness of requirements document”, “Structuredness of Code” and “Structuredness of the set of Test Cases”. Usually, the set of entities in such classifications (in this case requirements document, code, test cases) can change often during the time. Therefore, we recommend to exclude classifications from the quality models and use these classifications as additional input when creating the checklists and templates. There, classifications like failure types, available documents in the system development process and classification of changes (add, modify, delete) are supporting artefacts for the elicitation and documentation of the NFRs.

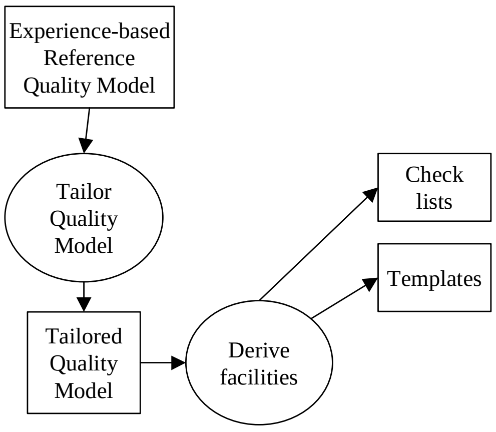


Figure 5-27 Tailoring of a quality model and facilities

Experience-based reference quality models are presented in Section 5.3.6. The quality models include quality attributes and their refinement, but no means and metrics to make the models more comprehensible. Furthermore, influence relationships between attributes are also not denoted in the graphical models. Where appropriate, the textual descriptions of these models reflect the means, metrics and influences. The organizational QAs that are listed in ISO 9126 (see [ISO01]), i.e., the compliance to standards, were also left out of the quality models, but applies for each high level quality attribute on the highest refinement level (as modelled in Figure 5-26).

5.3.2.4 Developing and adapting checklists and templates

Figure 5-27 describes the process of tailoring the quality model to the project and company. The tailored quality model is used as input to develop checklists and templates for documenting NFRs.

The structure of the checklists is given by the hierarchy of the quality models. General QAs (e.g., time behaviour) are, therefore, a means for structuring the checklist, while the QAs at the lowest level (e.g., usage time) are directly used to elicit the NFRs constraining them. The type of the QA influences the way the questions in the checklist are phrased:

- Organizational QAs are used in initialisation checklists that focus at general aspects in contrast to the concrete system or its task.
- User task QAs are iterated over the use cases (e.g., use case 1, then use case 2)
- System task QAs are iterated over the use case steps (e.g., step 1, then step 2)
- System QAs are iterated over the various subsystems in the system (e.g., database first, then network1)

When creating the checklists, one has to take into account the various types of classifications. E.g., for a checklist checking the maintainability QA, the classification of changes and the classification of physical products is important, as NFRs can be expressed on these classifications. For reliability, a classification of failures is important. Examples of classifications and of a checklist can be found in Appendix A and B.

One has to distinguish between checklists for the creation of the requirements document and checklists for the creation of the specification document. For the former ones, the customer QAs are mostly used to focus the checklists on the problem refinement. For the latter ones, developer QAs are used to refine the customer NFRs to a developer level. And means are suggested to make a further refinement possible. This refinement benefits, as most times customer NFRs are firstly measurable at the end of the product (e.g., response time), whereas a developer requirement can be measurable in earlier phases (e.g., modularity after creating the architecture).

The structure of the template is also strongly influenced by the quality model. As a result of the process “derive facilities” described above, the requirements template is created. Figure 5-28 shows a subset of this template. The template is part of the system requirements document and of the system specification document respectively. The NFRs constraining the different types of QAs are denoted at different places in the template:

- NFRs constraining the organizational QAs are documented in the section main constraints of the system (1.2.4) if they are stating organizational requirements like the size of the company, or the company’s experience. Process, Documentation, and stakeholder related requirements are documented in the Project Issues section (6.1, 6.2 and 6.3)
- NFRs constraining user task QAs are attached to the use case diagrams and are, therefore, documented in a use case diagram section (4.2.1).
- NFRs constraining system task QAs are directly attached to each use case in the textual use case description section. Therefore, the use cases have a field “NFRs”, where each system task oriented QA is listed (4.2.2). Below such a system task oriented QA, there is a list of the use case steps that express system tasks (e.g., response time: step2, step4). The NFRs for each system task are then expressed at this use case step (e.g., response time: step2 - “The system has to respond within 2 seconds”, step4 - “…”).
- NFRs constraining system QAs are denoted at two places in the template. First, if a NFR constrains a system QA of a subsystem (e.g., “the database has to store 100000 entries”) that is used in a use case, the NFR is attached to that use case (4.2.2). Therefore, each use case also includes a list of system QAs in the field NFRs. Below such a system QA, there is a list of all subsystems (e.g., capacity: database, memory). The NFRs for each subsystem are then expressed at this subsystem (e.g., capacity: database – “the system has to store 100000 entries”, memory – “…”). Second, the system NFRs are documented in the section of task overspanning NFRs (5). The structure is similar to the structure in the use cases (i.e., there is a list of all system QAs, below each system QA there is a list of all subsystems), but it aggregates the NFRs from all use cases and the ones that are not specific for one use case. Documenting the system NFRs in the Use Case is important to have all relevant information for the Use Case in its textual description. Documenting the system NFRs in the task overspanning part is done because a consolidation step searches for dependencies between NFRs concerning one subsystem.

1	Overview
1.2	Scope of the system
1.2.4	Main constraints
4	Functional requirements
4.1	Overview of system
4.2	Use Cases
4.2.1	Use Case Diagrams
4.2.2	Textual UC description
5	Non-functional requirements (task overspanning)
6	Project issues
6.1	Process stakeholders
6.2	Process requirements
6.3	Documentation requirements

Figure 5-28 Subset of the requirements document template

The template for the specification document template looks very similar, but has some developer specific elements. The specification template does not need to contain Use Cases (4.2), but notations used by developers to specify their view of the functionality of the system (e.g. statecharts). Means are not so relevant from a customers point of view, so they are not documented in requirements documents. From a developers point of view means are essential and have to be documented under the chapter architectural decisions, which is a subchapter of functional requirements (4).

An example of a requirements document template can be found in Appendix D.

5.3.3 The elicitation and documentation process

In the following sections, we describe the activities to be performed within the elicitation process. We use examples from a case study of the CWME project from Siemens [CS] about a wireless framework for mobile services. The application enables up to eight users to monitor production activities, manage physical resources, and access information within an industry plant. The user can receive state data from the plant on his mobile device, send control data from the mobile device to the plant components, position the maintenance engineer and get guidance to fix errors on machines. The case study is based on a real system and was provided by Siemens in the context of the Empress project.

5.3.3.1 Prerequisites

The elicitation process is based upon the documentation of

- the system's functionality (behaviour) described by use cases (UCs), used in the requirements document and by other notations (e.g. statecharts) used in the specification document,
- the physical architecture, if available, and further implementation constraints (e.g. constrained HW-resources or constraints derived from the operating systems),
- QA specific assumptions about the system in use. For certain QAs (e.g. Efficiency), assumptions about the average and the maximum amount of data used in the system are important, for others (e.g. usability) user models are important,
- various classifications like classification of change, classification of products, or classification of failures.

As described above, some of the QAs are associated to user tasks and system tasks. Therefore, we recommend use cases to describe the FRs from a customers point of view in the requirements document. This seems to be beneficial, because customer QAs associated to user tasks can directly be related to use cases. Customer QAs associated to system tasks can directly be related to use case steps. However, in other cases (e.g. developer QAs documented in specification documents) other notations supporting our conceptual entities, are used.

Figure 5-29 shows the pre-required information and the activities to create this information.

- Activities "Prioritise" and "Choose & Tailor quality models": Many times, budget and time limitations oblige to prioritise and focus on a subset of high-level QAs most important for a project. This activity is supported by a prioritisation questionnaire developed at IESE. It builds a ranking order for the QAs described in ISO 9126 (e.g., maintainability, efficiency, reliability, and usability). The questionnaire is described in more detail in Section 5.3.3.2. Based on this ranking order, quality models for certain high-level QAs relevant for the project can be chosen and tailored. The high level QAs that are most important shall be treated according to our approach to get a minimal, complete and focused set of NFRs. Of course there should also be some basic kind of elicitation for the minor important high level QAs (for example, just asking the stakeholders for NFRs concerning e.g., usability).
- Activity "Elicit functional requirements": In this step, the FRs are elicited and documented – in the requirements document in form of a graphical use case-diagram. Each use case included in the diagram is later associated to NFRs that constrain QAs of user tasks. In addition, each use case is described textually. The textual description includes an interaction sequence between actor and system. This description allows us later to associate NFRs that constrain QAs of system tasks to use case steps.

– in the specification document in a notation more suitable to specify the developers view of the systems functionality (e.g. statecharts).

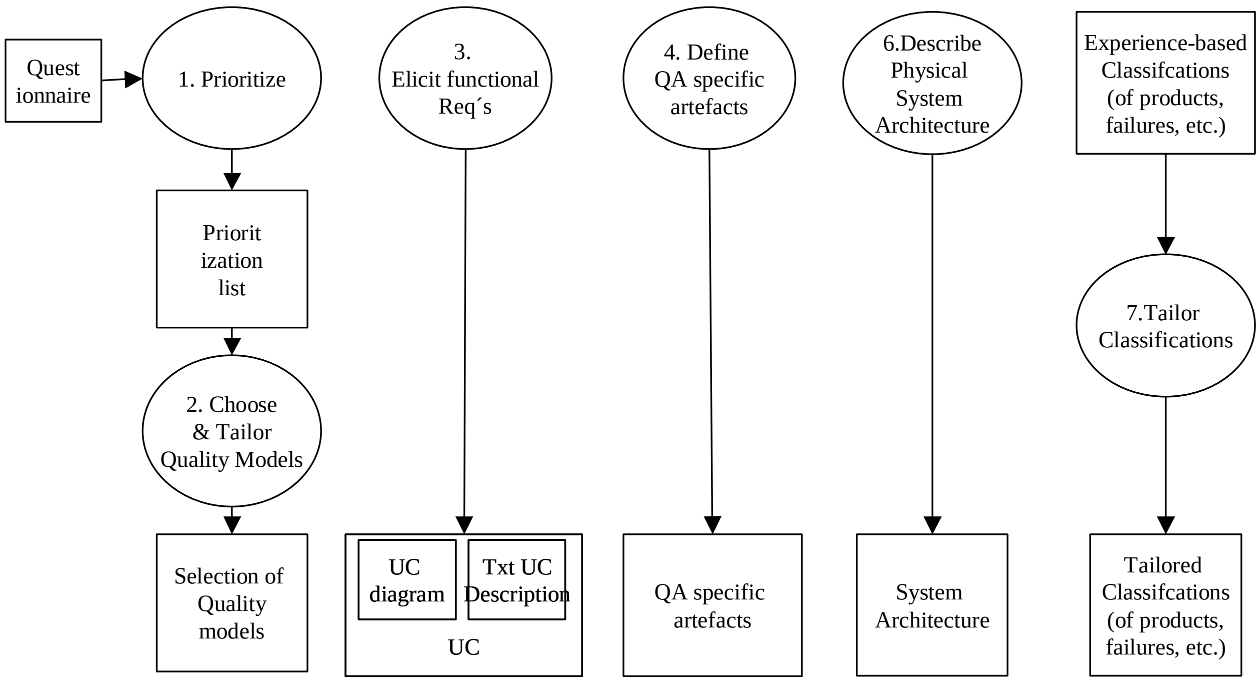

Quest ionnaire
1. Prioritize
Priorit ization list
2. Choose & Tailor Quality Models
Selection of Quality models
3. Elicit functional Req´s
UC diagram
Txt UC Description
UC
4. Define QA specific artefacts
QA specific artefacts
6.Describe Physical System Architecture
System Architecture
Experience-based Classifcations (of products, failures, etc.)
7.Tailor Classifications
Tailored Classifcations (of products, failures, etc.)

Figure 5-29 Development of prerequisites

- Activity "Define QA-specific artefacts": In order to be able to imagine NFRs, typical information related to a certain quality model must be captured. (e.g. efficiency quality model: maximum and average usage data; usability quality model: user model).
- Activity "Describe physical system architecture": Some NFRs can only be elicited if the detailed physical system architecture is known. So the architecture must be elicited and documented, whenever it is available. Again, the appearance of the architecture may vary, depending on the abstraction level of specification (system vs. software) and the type of quality model used (e.g. efficiency quality model: physical system components; usability quality model: interface components).
- Activity "Tailor Classifications": Experience-based classifications like classification of changes, classification of products, and classification of failures are tailored to the specific needs of the project and company. For example, a classification of failures will look different for a power plant development project than for the development of a spreadsheet application.

5.3.3.2 Prioritising Quality Attributes

For the prioritisation of quality attributes, an elicitation method is used. This method is based on a standardized questionnaire (see Appendix C) to elicit wishes and facts concerning the development context of the customer and relating them to a selection of quality attributes defined by ISO9126. The selection was chosen according to the research focus relevant for our project partner Siemens. They are: maintainability, efficiency, usability and reliability.

5.3.3.2.1 Development of the questionnaire

To develop the scales of the questionnaire, in a first step, potential scale items were generated. For this purpose, we phrased a large set of 120 statements containing wishes and facts, to which a person involved in SW development would agree. The statements covered the complete set of sub-characteristics of the high-level quality attributes mentioned above.

Once the statements were generated, they were presented to a selection of experts in software quality aspects in order to judge if a customer, that boosts the implementation of a certain quality aspect would agree to each statement. A 1-5 rating scale was used for the judgment. The experts were –as usual in scale development- not asked to rate their own project context, but rather to judge based on their personal experience, how favourable each item is with respect to the construct of interest.

In a next step, items with the highest mean and lowest variance (high interrater reliability) were selected and assembled to a 30 items questionnaire. As response scale, a 1-5 Likert scale was chosen, of which 17 statements covered facts of the current project (strongly agree – strongly disagree), and 13 statements covered wishes for future conditions (very important – very unimportant).

The determination of the mean value of the statements affecting one quality attribute enables to build a rank order of these. The one with the highest ranking is the most important attribute for the current software development project and should receive the greatest deal of attention. The latter is of special interest in case of limited requirements engineering resources and allows focusing on few activities.

The priorisation questionnaire and instructions for interpreting it can be found in Appendix C.

5.3.3.3 Eliciting, Documenting and Consolidating NFRs

The elicitation process is guided by our experience that different types of QAs address various entities (e.g., user task, system task). Each NFR has to be elicited under consideration of this entity. In addition, if an entity is described by one or a set of documentation elements (e.g., a user task is described by a use case, a system task is described by a step of a use case), the NFR has to be documented together with this entity.

In the following sections we describe the elicitation process for customer QAs documented in the requirements document.

Figure 5-30shows the activities and documents needed to elicit, document and consolidate NFRs from a customer's point of view in a requirements document. A checklist that is derived from the quality model as described in Section 5.3.2.4 guides each activity. We recommend using the checklist in a workshop-like fashion, i.e., one person either being a consultant or playing the role of a consultant who asks the questions on the checklist. Of course, it is very beneficial, if this person is knowledgeable in the elicitation process. Furthermore, all stakeholders that are addressed by the QA under discussion shall be involved in such a workshop. For example, the maintainer of a system shall be involved when talking about maintainability. A usage of the checklist by one person on its own is possible, but not recommended, as the aforementioned benefits can not be achieved in such a way.

In the following, the activities in Figure 5-30 are explained in more detail. We distinguish between different elicitation activities: user task NFR elicitation, system task NFR elicitation and system NFR elicitation. Each activity focuses on eliciting NFRs that constrain one certain type of QA (i.e., organization QA, user task QA, system task QA, and system QA). The user task NFR elicitation is based on use cases. The system task NFR elicitation is based on the interaction sequence described for each use case. The system NFR elicitation is based on physical subsystems and interaction sequences.

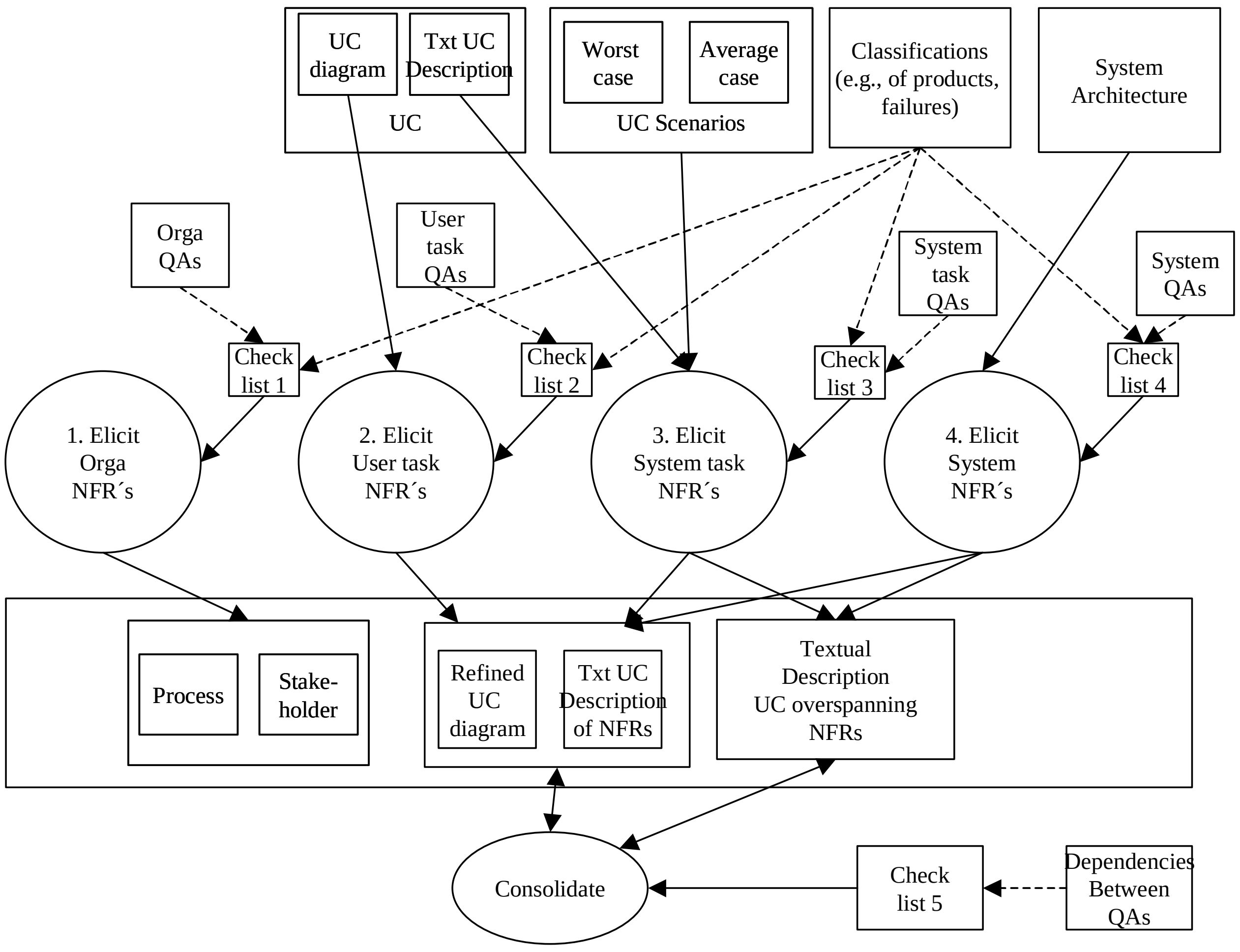


Figure 5-30 Elicitation, documentation and consolidation process for customer view NFRs

5.3.3.3.1 Activity “Elicit organizational NFRs (from a customers viewpoint)”

In this activity, NFRs that constrain QAs of the organization are elicited. The customer, for example, might have certain requirements concerning the organizational structure and experience of a supplier. The customer is asked to phrase these requirements. This process is guided by a set of clues in form of a checklist. These clues suggest thinking about domain-experience, size, structure or age of the supplier organization, as well as required standards (e.g. RUP), activities (e.g. inspections), documents or notations (e.g. statecharts). In our case study, some of the requirements expressed were:

- “The supplier needs at least three years of experience in the domain of access-control.“
- “The supplier has to create a specification document.”

To avoid unnecessary design limitations, the customer is instructed to scrutinize this NFR over and over again, just as Socrates used to try to get to the bottom of statements over and over. This form of Socratic dialogue serves to uncover the rationale behind that NFR and prevents the customer from constraining the system unnecessarily. NFRs are reformulated until they reflect the rationale. It is a good practice to document the rationale as well [DP01].

As soon as the now elicited and justified NFRs are phrased in a measurable way (this is the case if the metric attached to the QA in the quality model can be applied to the requirement), it is documented in the corresponding chapters of the template (see Section 5.3.2.4).

5.3.3.3.2 Activity "Elicit user task NFRs (from a customers viewpoint)"

In this activity, NFRs that constrain QAs of user tasks are elicited. In our case study, the QA "usage time" included in the quality model is a user task QA. These QAs are documented for each use case included in the use case diagram, because each use case represents a user task. As shown in Figure 5-31, NFRs are added to use cases with the help of notices.

In our case study the requirement "the use case shall be performed within 30 min." was attached to the use case "Handle alarm". Again, a justification as described above is performed to prevent unnecessary anticipated design decisions. The resulting rationale "breakdown of plant longer than 30 min. is too expensive" is documented in parenthesis behind the NFR.

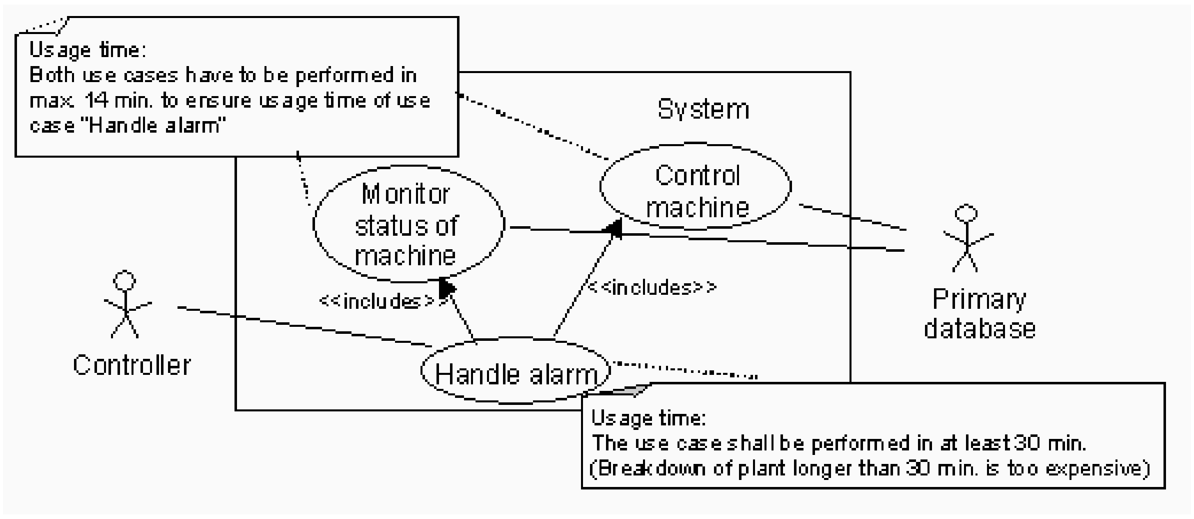
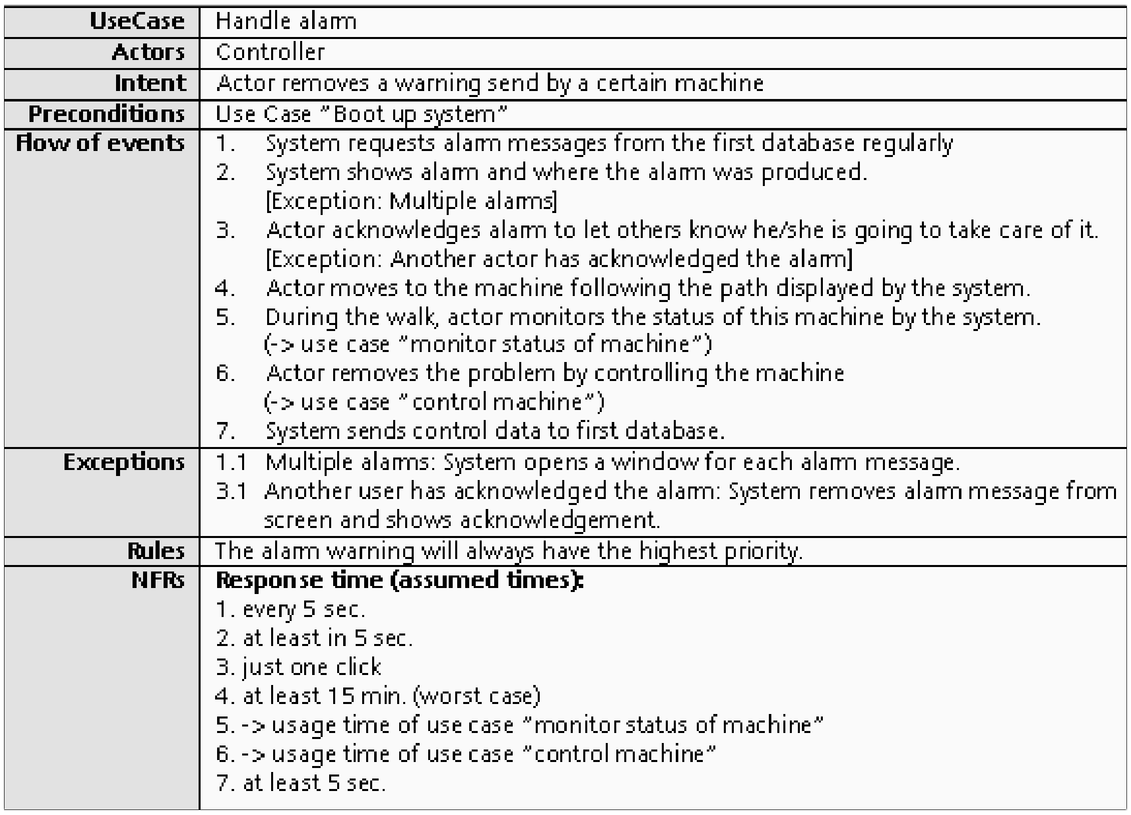

Usage time:
Both use cases have to be performed in max. 14 min. to ensure usage time of use case "Handle alarm"
System
Control machine
Monitor status of machine
<<includes>>
<<includes>>
Controller
Handle alarm
Primary database
Usage time:
The use case shall be performed in at least 30 min.
(Breakdown of plant longer than 30 min. is too expensive)

Figure 5-31 Use cases with attached user task NFRs

5.3.3.3.3 Activity "Elicit system task NFRs (from a customers viewpoint)"

In this activity, NFRs that constrain QAs of system tasks are elicited. The elicitation is based on the detailed interaction sequence (also called flow of events) documented in the use case. For this activity, quality model specific artefacts are needed. For the elicitation of NFRs based on the efficiency quality model, maximum and average usage data (Figure 5-29 shows the development process of this information) is needed. For the elicitation of NFRs based on the usability quality model, user characteristics are needed. The checklist gives clues of thinking of scenarios where this information is related to the system. With these scenarios in mind, every step and every exception described by the use case description are checked. Elicited NFRs are documented in the NFR field of the use case. Figure 5-32 shows the textual description of the use case "handle alarm". It describes that the system shows an alarm and where the alarm was produced. As reaction to this, the user acknowledges the alarm, so other users know s/he is taking care of it.



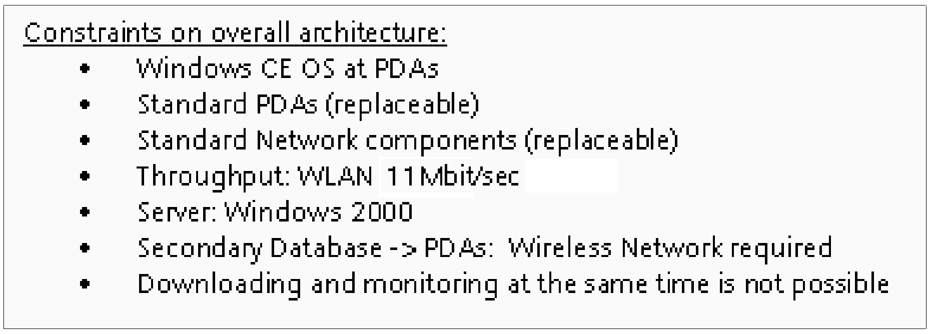
UseCase	Handle alarm
Actors	Controller
Intent	Actor removes a warning send by a certain machine
Preconditions	Use Case "Boot up system"
Flow of events	1. System requests alarm messages from the first database regularly 2. System shows alarm and where the alarm was produced. [Exception: Multiple alarms] 3. Actor acknowledges alarm to let others know he/she is going to take care of it. [Exception: Another actor has acknowledged the alarm] 4. Actor moves to the machine following the path displayed by the system. 5. During the walk, actor monitors the status of this machine by the system. (-> use case "monitor status of machine") 6. Actor removes the problem by controlling the machine (-> use case "control machine") 7. System sends control data to first database.
Exceptions	1.1 Multiple alarms: System opens a window for each alarm message. 3.1 Another user has acknowledged the alarm: System removes alarm message from screen and shows acknowledgement.
Rules	The alarm warning will always have the highest priority.
NFRs	**Response time (assumed times):** 1. every 5 sec. 2. at least in 5 sec. 3. just one click 4. at least 15 min. (worst case) 5. -> usage time of use case "monitor status of machine" 6. -> usage time of use case "control machine" 7. at least 5 sec.

Figure 5-32 UC steps with attached system task NFRs

As a result of the elicitation and documentation process, NFRs that constrain the system task QA "response time" were documented. The NFR "at least in 5 sec." was attached to the use case step 2 "System shows alarm and where the alarm was produced" and the NFR "just one click" was attached to the users reaction described in use case step 3. Both requirements were documented in the NFRs field within the textual description of the use case, after being justified by the customer in the Socratic dialogue. The rationale lead to the statement that the NFRs elicited were only assumed latency times and could be changed, if necessary. As shown in Figure 5-31 and Figure 5-32, the rationale was documented in parenthesis.

5.3.3.3.4 Activity “Elicit system NFRs (from a customers viewpoint)”

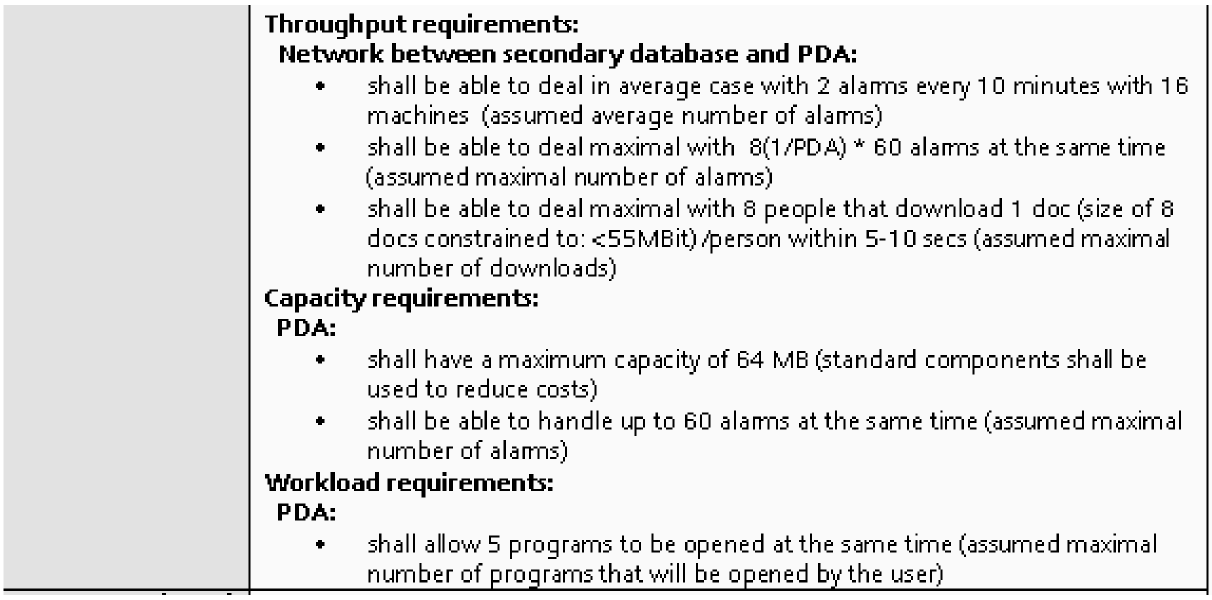
In this activity, NFRs that constrain QAs of the system and subsystems are elicited. In this activity, quality model specific artefacts are needed again. Additionally, the architecture (in case of NFRs derived from the efficiency quality model, the architecture contains physical subsystems) is used, if available. The subsystems and architecture constraints on our case study are shown in Figure 5-33.


Constraints on overall architecture:

- Windows CE OS at PDAs
- Standard PDAs (replaceable)
- Standard Network components (replaceable)
- Throughput: WLAN 11Mbit/sec
- Server: Windows 2000
- Secondary Database -> PDAs: Wireless Network required
- Downloading and monitoring at the same time is not possible

Figure 5-33 Constraints on system-architecture

The checklist gives instructions on how to consider the quality model specific artefacts while phrasing NFRs for each use case description and physical subsystem of the system architecture. As Figure 5-34 shows, the NFR field of the use case description is segmented into NFRs related to every physical subsystem.


Throughput requirements:

Network between secondary database and PDA:

- shall be able to deal in average case with 2 alarms every 10 minutes with 16 machines (assumed average number of alarms)
- shall be able to deal maximal with 8(1/PDA) * 60 alarms at the same time (assumed maximal number of alarms)
- shall be able to deal maximal with 8 people that download 1 doc (size of 8 docs constrained to: <55MBit) /person within 5-10 secs (assumed maximal number of downloads)

Capacity requirements:

PDA:

- shall have a maximum capacity of 64 MB (standard components shall be used to reduce costs)
- shall be able to handle up to 60 alarms at the same time (assumed maximal number of alarms)

Workload requirements:

PDA:

- shall allow 5 programs to be opened at the same time (assumed maximal number of programs that will be opened by the user)

Figure 5-34 UC with attached system NFRs

In the use case “handle alarm”, NFRs for the QA “capacity” could only be phrased for the physical subsystem “PDA”. The subsystem shall have a maximum capacity of 64 MB and shall be able to handle up to 50 alarms at the same time. The rationale for this NFR is the need for usage of standard components available at the consumer market. This rationale is documented as well.

The QA "throughput" does only apply to the subsystem "Network" by definition. Our experience shows, that some QAs are related to only a subset of subsystems. This relationship is documented in the quality model.

The elicited NFRs for single subsystems are documented within the textual use case description as well as in the section "use case overspanning textual description of NFRs". This is done to be able to consolidate the requirements over several use cases.

5.3.3.3.5 Activity "Consolidate (customer NFRs)"

In this activity, the NFRs are analysed for conflicts. The checklist 5 in Figure 5-30 gives hints on how to identify conflicts and how to solve them. This activity includes two sub-activities.

Checking for QA intra-dependencies:
In this first sub-activity, NFRs for one physical subsystem are analysed over all use cases. It has to be checked if NFRs on the same QA are redundant or in conflict. E.g., on one use case, one might specify that the wireless network must be 11 Mbit/s, whereas an NFR on a second use case was stating that the wireless network has to support 55Mbit/s. As all NFRs to one system QA are documented in the task overspanning NFR section of the template, it is easy to check for such redundancy and conflicts. The NFR for such a system would then be documented as "The throughput of the wireless network must be at least 55 Mbit/s". As the check needs the system QAs to be completely elicited, this consolidation step can be conducted as soon as all NFRs to one QA are elicited. We recommend performing this step after all NFRs have been elicited, as NFRs can appear and change until the end of the overall elicitation phase.

Checking for QA inter-dependencies:
In the second sub-activity, NFRs that constrain different QAs are validated under consideration of the dependencies documented within the quality models. Checking for these dependencies is a non-trivial task. One can think of many alternative ways for performing this analysis. In the following, we explain the various alternatives by way of example.

The checklists in the activities "elicit …NFRs" (see Figure 5-30) are used to elicit different NFRs related to the QAs in the quality model. The questions in the checklist are related to QAs in the quality model (see Section 5.3.2.4). In other words, one or more questions lead to NFRs related to a QA in the quality model.

The following is an extract of such a list of NFRs related to different QAs of the quality model. See the results of the demonstrator of the Empress case study provided by Siemens for a detailed list of NFRs regarding the different QAs of the quality models.

Efficiency requirements:

Boot Time requirements:
R.B.1 The PDA shall boot up in 5 seconds.

Capacity requirements:
R.C.1 The PDA shall have 64MB of RAM.

Workload requirements:
R.W.1 It must be possible to open 5 programs at the same time.

Reliability requirements:
Recover time requirements:
R.RT.1 The PDA has to recover within 5 seconds.

The experience-based quality models indicate the following influence dependencies:

Reliability: Recover time ⇔ Efficiency: Boot time

Efficiency: Capacity ⇔ Efficiency: Workload

As the QA have influence dependencies, it is possible, but not mandatory that also the NFRs, expressed over the QAs have dependencies. When looking closer at the different NFRs there are two influence dependencies between the NFRs that can be identified:

- The PDA has to recover in 5 seconds – The PDA shall boot up in 5 seconds.

  Since the recover process may include a boot up process and some reestablishment of the system state, the time given for the recover process may be too low.

- The PDA shall have 64 MB of RAM – It must be possible to open 5 programs at the same time

  The 64MB of RAM influence the amount of programs that can be opened at the same time. Maybe 64 MB are not enough.

For discovering of such influence dependencies between two NFRs, a process shall be defined that helps. In the following, different possibilities to identify influence dependencies between NFRs will be introduced. The alternative processes are evaluated regarding effort and accuracy aspects. Accuracy aspects regard the number of influence dependencies that can be discovered using a specific process.

To be able to evaluate the different processes the following variables are used:

Variable	Description	Estimated values in Case study
N	Amount of NFRs elicited	100
g	Number of NFR groups (a group relates to a QA the NFR have been elicited from)	20
n	Number of NFRs per group (in practice this number is not the same for each group, for the example this is an estimated average value)	5
d	Number of influence dependencies identified in the quality model between QAs	7

The row to the right relates to the case study conducted, it indicated estimated values for the different variables that will be used to assess the processes.

The different alternatives that can be distinguished are:

Alternative 1

Process steps

1. For each NFR, check if it has an influence on all other NFRs.
2. If there is an influence write down this relationship.

Effort

The number of comparisons is:

$$(N-1)+(N-2)+...+1=\frac{N\cdot(N-1)}{2}$$

For the example this would be:

$$\frac{(100)\cdot(100-1)}{2}=4950$$

This effort is very high since no experience of the quality model is used to determine influence dependencies.

Accuracy

Since all NFRs are compared, all influence dependencies can be found. The accuracy is in the best case 100% (depending on the experience and perception of the person conducting the comparisons).

Alternative 2

The NFRs will be grouped according to the related QAs and the influence dependencies identified in the quality model will be used to compare.

The NFRs are grouped according to the QAs they were elicited from. The groups are the QAs of the quality models. This grouping process is already done while eliciting the NFRs. First, the checklist questions that were derived from the QAs result in an already grouped set of NFRs. Second, the template used to document the NFRs reflects these groups. Therefore, the grouping process (step 1) described below is conducted implicitly during the elicitation process.

The number of NFRs per group should be limited to a certain value. If the number of NFRs per group exceeds a certain maximum value, the NFRs should be grouped more fine grained, because the effort (number of comparisons) and the complexity (number of NFRs that should be present in mind) of a dependency check based on these groups increases as the number of NFRs per group increases.

One possibility could be to further group the NFRs related to a QA, according to the type of QA.

- For a user task or system task QA: Group the NFRs according to the use cases they are elicitated from.

  E.g.: Availability is elicitated over three different Use Cases UC1,UC2 and UC3. The resulting groups could then be UC1, UC2 and UC3 with all NFRs.

- For a system QA: Group the NFRs according to the architectural object they refer to.

  E.g.: There are two NFRs for response time, one for a PDA and one for a database, so there could be two groups of NFRs PDA and database.

Process steps

1. Group the NFRs according to the QA in the quality model they were elicited from. (given by the template and checklist)
2. For each group, check if the related QA in the quality model has a influence dependency to another QA in the quality model
3. If there are influence dependencies to other QAs
   a. For each related group of NFRs, check if there are influence dependencies between the NFRs.

b. If there is an influence dependency, write down the relating NFRs.

Process step 3.a is expressed in a very general way. In the following, two different ways to check for influence dependencies between two groups of NFRs are presented in more detail.

- Check each NFR of one group with each NFRs of the second group
- Check the first group of NFRs with the second group of NFRs

Alternative 2a

Changed process step:

a. For each NFRs of the first group check for a influence dependency to each NFR in the second group.

This alternative is similar to Alternative 1, but conducted within a group, not on the complete set of NFRs.

Effort

The number of comparisons now only relates to the different groups. To compare two groups to each other the amount of comparisons to be conducted will be:

$$(n-1)+(n-2)+...+1=\frac{n\cdot(n-1)}{2}$$

For the example this would be for one group:

$$\frac{(5)\cdot(5-1)}{2}=10$$

The overall number of comparisons is then

$$7*10=70$$

The effort is essentially lower than the effort for alternative one (4950 comparisons). In practice, this number depends on the size of the project and can be significantly higher because the number of NFRs per group n, is higher.

Accuracy

The accuracy in comparison to alternative 1 is usually lower, because only influence dependencies between NFRs of groups are found, that have an influence dependency in the quality model. Thus the accuracy depends on the quality of the influence dependencies in the quality model. If all influence dependencies are indicated in the quality model then all influence dependencies between NFRs can be found.

Alternative 2b

Changed process step:

a1. Read through all NFRs of the first group.

a2. Read through all NFRs of the second group.

b. Write down the influence dependencies identified between the two groups.

Effort

Since only the groups are compared to each other there is one comparison per group, so in total d comparisons, in the example 7 comparisons. The effort is lowest of all alternatives.

Accuracy

The accuracy is comparable to alternative 2a but since only the complete groups of NFRs are compared to each other it is usually lower.

Summary and Recommendation

The different alternatives have advantages and disadvantages regarding effort and accuracy. A trade-off has to be made between these two aspects. Since the effort regarding alternative 2 (2a and 2b) is essentially lower than that of alternative 1 and the accuracy in comparison to alternative 1 is at least acceptable (assuming a good quality model with regard to influence dependencies), alternative 2 is the preferred alternative of those two.

A choice for alternatives 2a and 2b depends on the number of NFRs per group:

- If the number of NFRs per group is fairly low, alternative 2a should be used, as the overall effort for checking for influence dependencies is fairly low and the accuracy is higher.
- If the number of NFRs is higher alternative 2b should be used as the effort is too high.

When to check for influence dependencies

A very important aspect regarding the different processes is the point in time chosen to check for possible influence dependencies.

There are two possibilities:

1. Integrate the influence dependency check process in the elicitation process.
2. Define influence dependency check process as a standalone process that will be carried out after the elicitation process.

Alternative 1

Immediately after eliciting a NFR there will be an influence dependency check (according to one of the processes above). On the one hand, since the customer just elicited the NFR the knowledge regarding this NFR will be present, on the other hand the elicitation process is interrupted which could lead in possible quality loss regarding the elicitation of the NFRs.

Another major disadvantage regarding this alternative stems from the usually not linear process of elicitation. The questions of the checklist in the elicitation process are grouped according to the QAs in the quality model. The NFRs are gained successively according to the QAs. This process is not entirely linear in practice though, sometimes while eliciting NFRs related to a specific QA, customers think of other NFRs related to QAs that were already elicited at an earlier time.

For process alternative 1 there will be no difference in effort. The influence dependency check always compares the NFR elicited to all other NFRs so the point in time at which the NFR is elicited does not make a difference for the influence dependency check process.

For process alternative 2 though there is a major difficulty to handle. If at a certain point, while eliciting NFRs related to a QA, the customer discovers a NFR that belongs to a group of NFRs that has been elicited before, the influence dependency check for this group would have to be repeated. For example, let’s assume that there are 4 groups of NFRs, A to D. The NFRs related to group D are now elicited. Since the influence dependency check takes place immediately after the elicitation of a NFRs the influence dependency check for all NFRs of Group C with those of group A and B already took place while eliciting the NFRs of group C. If the customer now thinks of another NFR in group C, this influence dependency check would have to be processed again.

Alternative 2

If the influence dependency check is defined as a standalone process, the customer would have to recall the knowledge regarding the NFRs which is more difficult compared to alternative 1. On the other hand, neither the elicitation process nor the influence dependency check process would have to be interrupted which leads to better process results. Furthermore, multiple comparisons for influence dependencies can be anticipated, due to the fact the elicitation process is terminated and all NFRs are elicited at that time.

Summary and recommendation

In practice, it seems more pragmatic to use alternative 2, i.e., a standalone consolidation process for performing the consolidation step. Therefore, as depicted in Figure 5-30, we recommend a standalone consolidation step.

In the following section we describe the elicitation process for developer QAs documented in the specification document. Figure 5-35 gives an overview of the elicitation process. The activities are described in detail below. As this elicitation process is similar to the elicitation process for customer QAs in the requirements document, we focus on the differences to the activities described in the previous sections. Apparently, the elicitation of organizational NFRs is missing in the developer view. Typically, organizational requirements are expressed in the requirements document and seldom in the specification document. Another change that can be seen in Figure 5-35 is that means which are attached to the quality attributes are also an input to the elicitation activities.

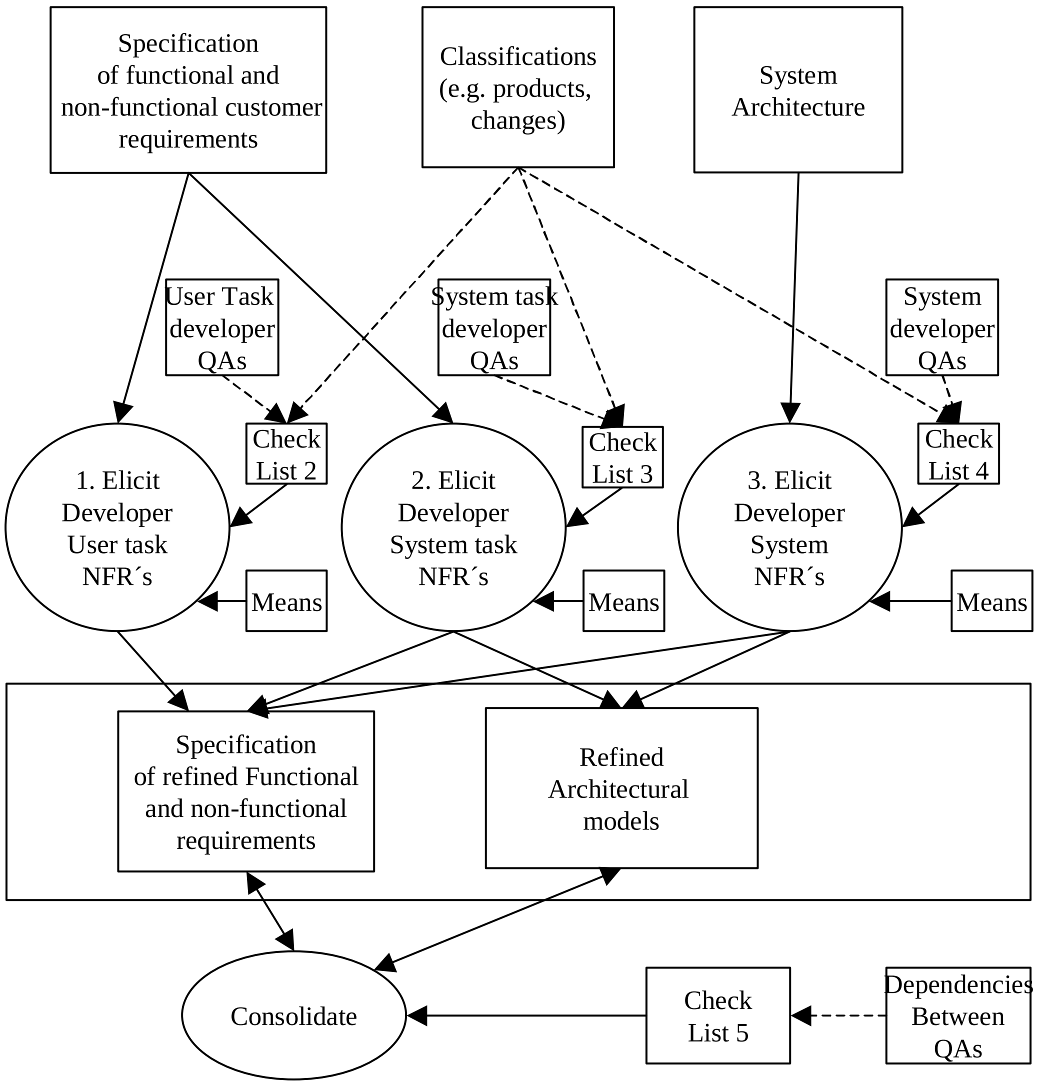

Specification of functional and non-functional customer requirements
Classifications (e.g. products, changes)
System Architecture
User Task developer QAs
System task developer QAs
System developer QAs
Check List 2
Check List 3
Check List 4
1. Elicit Developer User task NFR´s
2. Elicit Developer System task NFR´s
3. Elicit Developer System NFR´s
Means
Means
Means
Specification of refined Functional and non-functional requirements
Refined Architectural models
Consolidate
Check List 5
Dependencies Between QAs

Figure 5-35 Elicitation process for developer view QAs

5.3.3.3.6 Activity "Elicit developer user task NFRs"

In this activity, NFRs that constrain QAs of user tasks from a developers point of view (compare Section 5.3.3.3.2) are elicited. Therefore the specification of functional and non-functional customer requirements is needed. The quality model contains so called "means". Means contain possible solutions to requirements derived from a QA. These means capture the experience of developers and are documented in the specification. Further, to reduce the complexity of the process, specific classifications (e.g. products, changes) are used to phrase NFRs on a more detailed level.

5.3.3.3.7 Activity "Elicit developer system task NFRs"

In this activity, NFRs are elicited that constrain QAs of system tasks. The elicitation is based on the detailed behavioural description. For this activity, quality model specific artefacts (Figure 5-29 shows the development process of this information) are needed (compare Section 5.3.3.3.3).Means will also influence the documentation of system task NFRs. Again, quality model specific classifications (e.g. products, changes) are used to phrase NFRs on a more detailed level.

5.3.3.3.8 Activity “Elicit developer system NFRs”

In this activity, NFRs that constrain QAs of the system and subsystems from a developers point of view are elicited. In this activity, again quality model specific artefacts are needed. Additionally, the architectural description is used, if available (compare Section 5.3.3.3.4). Again, quality model specific classifications (e.g. products, changes) are used to phrase NFRs on a more detailed level. The use of means to make a further refinement possible (see Section 5.3.2.3) can have an impact on the architectural description (e.g., when using design patterns as a means).

5.3.3.3.9 Activity “Consolidate (developer NFRs)”

The consolidate activity differs slightly from the consolidate customer NFRs activity of Section 5.3.3.3.5. The difference is that choosing a means as a solution can have huge impact on other high level QAs. We recommend that the developer shall document such means with an assessment matrix for each subsystem under consideration (see Table 1). The rows of the matrix represent the selected means. The columns of the matrix represent the requirements that are relevant to the subsystem under consideration and the means have an impact on. Each cell denotes whether a specific means makes it easier or more difficult to realize the corresponding requirement with the symbols “+” and “-“ and a reference to the scenario that was used to generate the value. If the means has no impact on the requirement, the cell is left empty.

	FR1	FRn	Efficiency	Maintainability
Locality			-	+
Load balancing			+	-
Caching			+	-
Concurrency			+	-
Sharing			+	-

Table 5-11. High-level assessment matrix for detecting conflicts

Once the matrix has been filled out, the designer identifies potential conflicts between selected means and between means and quality attributes and can resolve them by choosing the best means, standing in the least conflicts with the existing NFRs.

5.3.4 Overview of the Quality Model

In order to success when dealing with NFR it is necessary to define a quality model to explicitly **identify, describe and relate** to each other these quality attributes. Plus, the quality model has to provide the developer with an explicit **way of analyzing** NFRs of a particular software system, and with a **methodology to validate** the software architecture from the NFR point of view.

The quality model basics are mostly based on the ideas and concepts presented on [Chu00].

5.3.4.1 Process and Activities

The quality model attempts to support some important aspects of NFR treatment:

NFR Identification and Description. Defining, describing and classifying NFRs are not easy tasks. As quality, it is difficult to say what they exactly mean, but it is easy to notice when they are missing. Different authors have proposed different definitions and different classification criteria in the recent years, but none of them has come to be a widely accepted standard. ISO/IEEE has also proposed a Quality Model in order to set up a common framework for NFRs (ISO Standard 9126), but it does not seem to have had a universal acceptance. Nevertheless, we will use it for the quality model basics, since it is the only “official” standard out there.

NFR Analysis. It is totally useless to be able to identify NFRs in software development if afterwards you don’t know how to deal with them. It is necessary to provide an analysis process to decompose NFRs from the system domain point of view (Security does not mean the same on an elevator-controlling software than on an online-banking embedded application) so that the requirements are split into smaller and more objective design goals. This analysis procedure is supported by a special tree-based notation. Since it is impossible to cover all the possible system domains (they are infinite!), the quality model provides the reader with some common analysis examples and some general rules that can be applied no matter the system domain.

5.3.4.2 Notation

As already said, the quality model notation and the quality attributes analyzing methodology is described on [Chu00]. The way to come up with such quality models is described in Section 5.3.2.3 and Section 5.3.3 shows how to use them for elicitation, documentation and consolidation of NFRs.

Figure 5-36 illustrates the basic quality model notation using a basic example.

The advantage of this notation is that it represents the relationships between quality attributes and design decisions and other quality attributes explicitly. Quality attributes are represented as “softgoals” (in the remainder of this chapter, “quality attribute” and “softgoal” will be used synonymous), which are depicted as solid-line clouds. These softgoals are incrementally refined, and trade-offs are made as conflicts and synergies are discovered. This goal-oriented design process is supported by an interdependency graph of softgoals. “Performance”, “Cost” and “Space” are examples of softgoals on Figure 5-36.

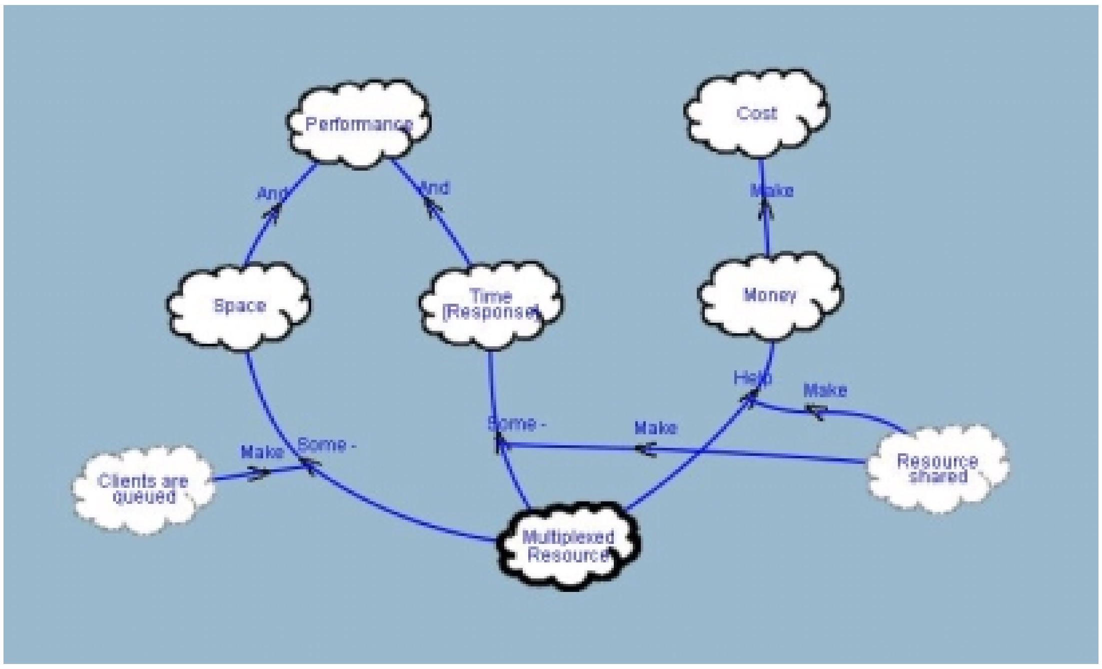


Figure 5-36: OME's notation example

At some point, when the NFRs have been sufficiently refined, one will be able to identify possible development techniques for achieving these NFRs. These development techniques that link softgoals to a particular design are called **Operationalizations**, and are represented as clouds with a thick solid border. The concept of means (see section 5.3.2.2 is almost the same as operationalizations, but is not limited to development techniques. For the remainder of this chapter, "means" and "operationalizations" will be used synonymous. "Multiplexed Resource" is an operationalization on Figure 5-36.

As already mentioned, softgoals are interdependent. The solid links between softgoals indicate interdependency between them. There are two types of interdependencies:

Refinement: Refinements can be "**And**" or "**Or**". They indicate, respectively, that all subgoals must be achieved for the parent goal to be achieved, or that achieving any one of them is sufficient. Figure 5-36 shows that in order to achieve the softgoal "Performance" it is necessary to achieve the softgoals "Space" and "Time Response" ("And" refinement).

Contribution: There are positive and negative contributions.

Positive contributions are "**Make**" (it means that the goal sufficiently contributes to the parent goal), "**Help**" (it indicates a positive contribution, but not sufficient on its own), and **Some+** (which says that there is some positive contribution to the parent goal, but it is weaker than Help).

Negative Contributions are "**Break**" (if the offspring is satisfied, the parent can be sufficiently denied), "**Hurt**" (if the offspring is achieved, partial negative support is given to the parent), and "**Some-**" (which represents some negative contribution, but it is weaker than Hurt).

These two types of interdependencies have been already introduced in the section 5.3.2.3. In that section the "Refinement" relationships are called "Refine" and what is called "Contribution" in this section includes "Influence (Means)" and "Influence(QA)" of section 5.3.2.3.

Other elements of the notation are **claims** and **correlation links**. Claims justify the choice of a contribution type on a link, and are shown as dotted-line clouds. Correlation links indicate side-effects that are not at the centre of our analysis. They are shown as dotted-line links. “Resource Shared” is an example of claim on Figure 5-36.

The interdependencies are used for NFR qualitative validation. They help propagate the degree of achieving softgoals through the contribution links using a qualitative reasoning process. The propagation rules are based on mathematical logic, and are explained in deep detail on the book. We are not going to address them here since the OME tool fully supports automatic qualitative achievement propagation, and it is not necessary to understand them to perform NFR validation.

The other elements that are needed for NFR qualitative validation are “**Labels**”. Labels are used to accept or reject design alternatives, and through the qualitative reasoning process the user can determine the impact of the decisions on other softgoals, and ultimately, upon the top quality attribute.

There are 6 types of labels:

X : Denied

W- : Weakly Denied

U : Undecided

W+ : Weakly Satisfied

√ : Satisfied (or accepted if applied on operationalizations)

↯ : Conflict

Figure 5-37 shows an example on the use of labels and NFR qualitative validation. It depicts how the quality attributes “Performance” and “Cost” could be fulfilled if the design solution “Multiplexed Resource” were used.

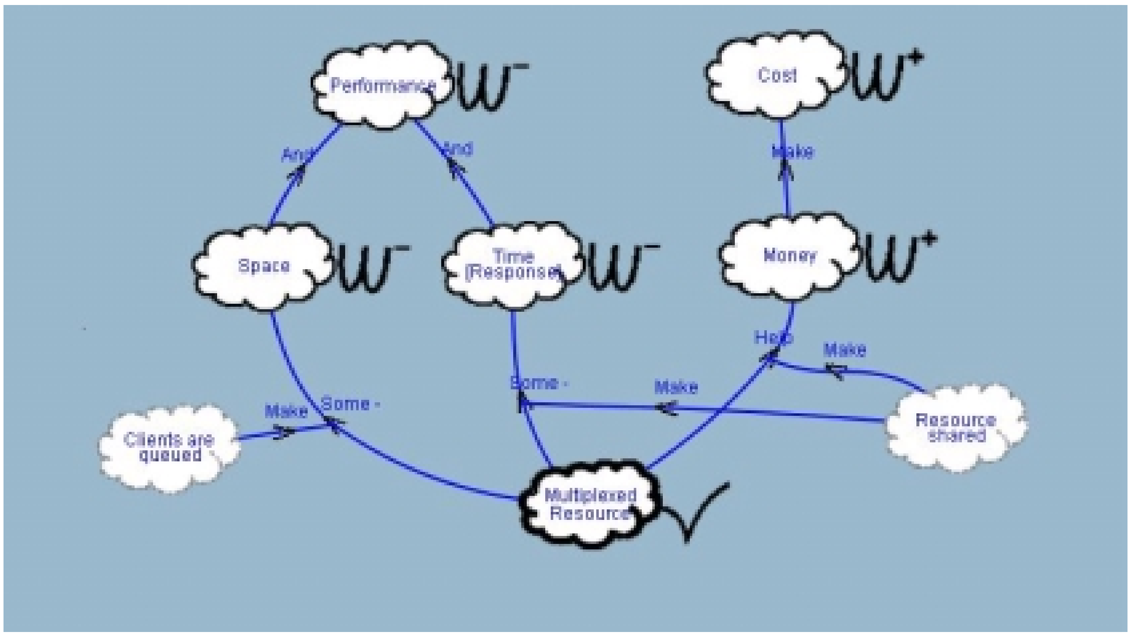


Figure 5-37: OME’s qualitative evaluation

After accepting “Multiplexed Resource” as applied operationalization (by adding the √ label), the tool automatically generates the rest of the labels. The final interpretation is that whereas the “Cost” requirement would be partly fulfilled, “Performance” would be partly denied. This matches the background idea of multiplexing: you save money by sharing the medium, but the performance is therefore jeopardised.

5.3.4.3 The OME tool

The OME (Organization Modelling Environment) tool is a small but powerful application which is being developed at the Knowledge Management Lab at the University of Toronto as a part of the "Agent-Oriented Approach to System Architecture: Models and Analysis Tools" project, funded by Communication and Information Technology Ontario with support from Mitel Corporation.

The current version of OME is 3.1, and it was developed in Java 1.2. It can be downloaded for free at http://www.cs.toronto.edu/km/ome/ for non-commercial purposes only. The tool supports different frameworks that have been developed by the University of Toronto, but we will focus only on the NFR one.

Installing and running the tool is rather automatic (sometimes it is necessary to specify where the Java Runtime Environment files are installed is the tool cannot detect them by itself). On the main window (**Figure 5-38**) click on “New Project” and type a name for it.

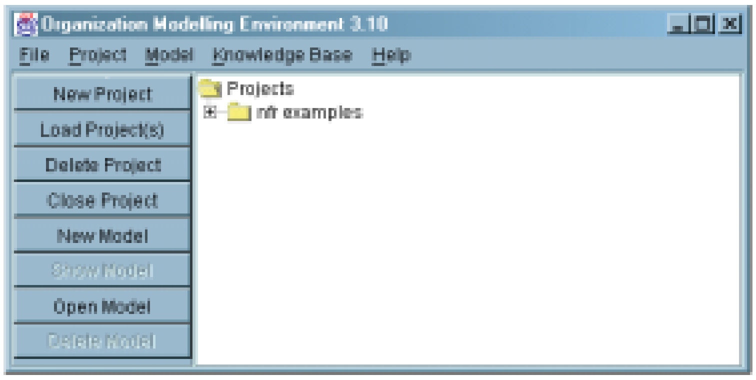


Figure 5-38: OME’s main window

Click now on “New Model”. Type a name for the new model on the new window, select the project that you have created a couple of minutes ago to place the model into it, and check the NFR checkbox to select the NFR notation. A new window will pop up to start modelling your quality attributes. The modelling windows will present different toolbars depending on the selected checkboxes. In order to make it easier to work with make sure that just the NFR checkbox is checked.

The tool is really simple to work with, being the only problem the lack of an online-help function (the “Help icon does not work” in version 3.10). To add a new item, just select it on the upper icon bar and then click on the place where you want to put it. In case you want to add an interdependency link between softgoals, select the link on the icon bar and then make two clicks: one on the origin and another on the destination cloud. Before displaying the link, you will be requested to select the type of interdependency you want to create (Make, Hurt, etc).

An element can be moved in the graph by clicking on the element and dragging the mouse while keeping the mouse button pressed. Links are always drawn as a straight line from their destination to their source and cannot be moved (except by moving the link’s destination or source). To delete an element, just select it and press the DELETE key.

A really useful contextual menu is available by selecting an element and making right click with the mouse. **Figure 5-39** shows for example how to set up a “NFR Satisfied” label (√)

on a particular softgoal. If the option “NFR Options->Autopropagation” is selected, by adding such a label the tool will automatically start the validation propagation process up through the sofgoal hierarchy. For each “cloud” on the hierarchy tree the tool will calculate the propagated value and show a window asking the user for confirmation. If you don not agree with the calculated value there is an option to manually set up the label you think is better.

In OME it is possible to **save** and **export** your model. Saving refers to storing a copy of your model on disk in order to use the model with OME later on. It is possible to save the model in three different models: in “tel” (the native OME format), “sml” or “xml”.

Exporting refers to saving a picture of the graphical representation of your model that can be used in other documents. Exported files are stored in .png picture format files. Any standard graphics application should be able to manipulate these images, and convert them to any other standard picture format. .png files can easily be included in word processor documents, or in web pages.

For more information about OME, please refer to the documentation of the tool.

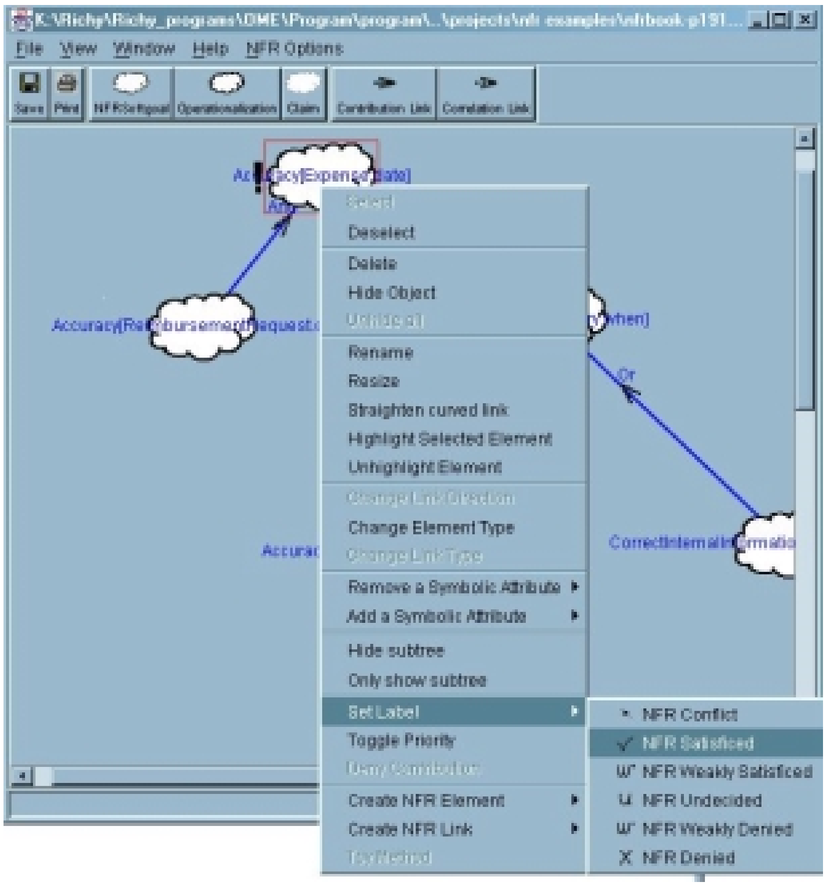


Figure 5-39: Setting labels with OME

5.3.5 Defining a NFR Base Force Hierarchy

5.3.5.1 ISO 9126 Standard

In order to analyse non-functional requirements in a proper way (that is, specify them measurably, identify inconsistencies, study relationships, etc) it is necessary to agree on what non-functional requirements are and create a Quality Model. Instead of creating our own Quality Model from scratch, we use the one defined in ISO/IEC 9126 (with some extensions from IEEE 1061) in order to have a globally and internationally accepted standardizing framework.

ISO/IEC 9126 provides a general-purpose model that defines three different quality areas:

Internal Quality Requirements: They specify the level of requirement from the internal point of view. They are used to specify properties of interim products which can include static and dynamic models or source codes among others. It corresponds to the "Developer's View".

External Quality Requirements: They specify the level of quality from the external point of view. It corresponds to the "Manager's View". A manager is typically more interested in the overall quality rather than in a specific quality characteristic, and for that reason, he will need to assign weights to the individual characteristics in order to reflect business requirements.

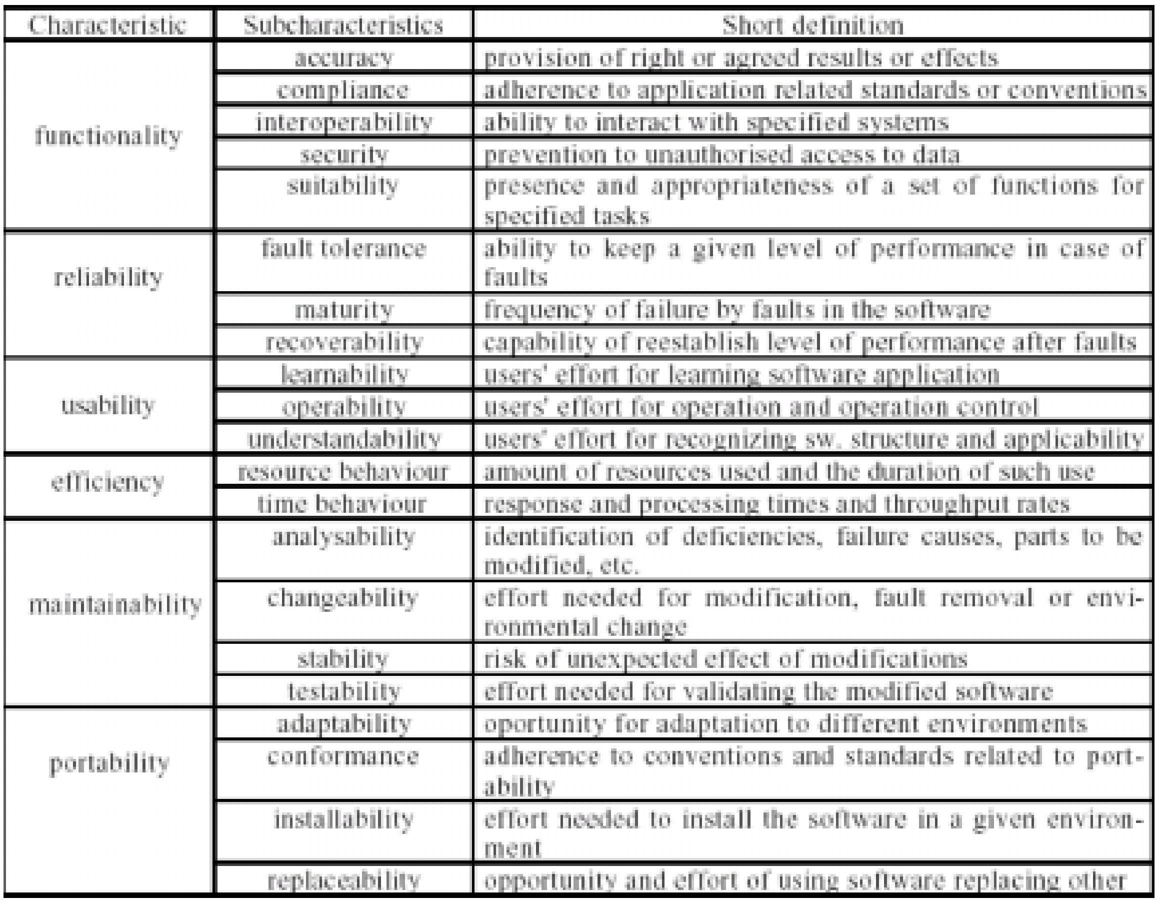
Quality in Use: It is the user's view when the product is used in a specific environment and in a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself.

For the first two areas, ISO/IEC 9126 defines six broad categories to quality: *functionality, reliability, usability, efficiency, maintainability and portability*. These provide a convenient classification structure (such as provided in a checklist) and must not be taken as indicating relative importance. The six categories are further broken down into sub-characteristics with measurable attributes.

For the third one, the standard defines 4 new categories: *Effectiveness*, *Productivity*, *Safety*, *and Satisfaction*.

The table on Figure 5-40 shows the ISO/IEC 9126 decomposition.

The following sub-chapters define and describe the categories and subcategories briefly. In order to make them easier to understand, we will show how they should be interpreted in the context of one of the simplest software products: a web page presenting information about a certain software company.



Characteristic	Subcharacteristics	Short definition
functionality	accuracy	provision of right or agreed results or effects
	compliance	adherence to application related standards or conventions
	interoperability	ability to interact with specified systems
	security	prevention to unauthorised access to data
	suitability	presence and appropriateness of a set of functions for specified tasks
reliability	fault tolerance	ability to keep a given level of performance in case of faults
	maturity	frequency of failure by faults in the software
	recoverability	capability of reestablish level of performance after faults
usability	learnability	users' effort for learning software application
	operability	users' effort for operation and operation control
	understandability	users' effort for recognizing sw. structure and applicability
efficiency	resource behaviour	amount of resources used and the duration of such use
	time behaviour	response and processing times and throughput rates
maintainability	analysability	identification of deficiencies, failure causes, parts to be modified, etc.
	changeability	effort needed for modification, fault removal or environmental change
	stability	risk of unexpected effect of modifications
	testability	effort needed for validating the modified software
portability	adaptability	oportunity for adaptation to different environments
	conformance	adherence to conventions and standards related to portability
	installability	effort needed to install the software in a given environment
	replaceability	opportunity and effort of using software replacing other

Figure 5-40: ISO NFRs classification

5.3.5.2 Functionality

Functionality is the set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. This set of attributes characterizes what the software does to fulfil needs, whereas the other characteristics mainly define when and how it does so. Put it into other words, the requirements related to this characteristic try to answer the two following questions:

Are the required functions available in the software?

Does the software satisfy the defined needs?

It comprises five different sub-characteristics: accuracy, compliance, interoperability, security and suitability.

Accuracy

According to ISO 9126, *Accuracy* is composed of those attributes of software that bear on the provision of right or agreed results or effects. Generally speaking, *Accuracy* is the presence of correct and predictable results from a specified input.

In the company web page context, *Accuracy* will apply to the information presented on the web: the information must correspond exactly with the tools, methods, products and training provided by the given company.

Compliance

ISO 9126 defines *Compliance* as the software quality characteristic that is composed by attributes that make the software adhere to application related standards or conventions or regulations in laws and similar restrictions.

Interoperability

Interoperability is defined by ISO 9126 as those attributes of software that bear on its ability to interact with specified systems. It indicates how easily the system can exchange data or services with other systems. To asses the required degree of interoperability it is necessary to know which other applications the users will employ in conjunction with the system in development, and what data they expect to exchange.

In the company web page context, *Interoperability* will apply to the possible software programs that might interact with the web site: Compatibility is required with Netscape 3 and 4, and Internet Explorer 3 and 4, and for downloading Acrobat Reader 3. Monitor sizes beyond 640x480 and 256-colours must not be required. All pages must be printable on low-cost colour inkjet printer and on postscript laser printers. Also it must be useable by people who are visually impaired and are using programs that read the screen contents in sequential order.

Suitability

ISO 9126 defines *Suitability* as the software attributes that bear on the presence and appropriateness of a set of functions for specified tasks.

In the company web page context, *Suitability* will address topics like “All information on the site must relate to *activities* of the company. Especially items relating to the company's views of the world, history or economic results are not suitable”.

Security

Security is defined by ISO 9126 as the software attributes that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data. It is important not to confuse *Security* with *Safety*, which is defined as a characteristic of Quality in Use[2] and does not relate to the software alone but to the whole system.

In the company web page context, *Security* will cover scenarios such as “For the download section, a login and a password will be required so that only registered and authorized people can download program trial versions or further documentation”.

5.3.5.3 Reliability

Reliability is the set of attributes that bear on the capability of software to maintain its level of performance under certain conditions for a period of time. It is also normally described as the probability of the software executing without failure for a specific period of time. Using a more precise definition we could say that *Software Reliability* is comprised of two activities:

Error prevention

Fault detection and removal

The terms errors, faults and failures are often used interchangeably, but do have different meanings. In software, an error is usually a programmer action or omission that results in a fault. A fault is a software defect that causes a failure, and a failure is the unacceptable departure of a program operation from program requirements. When measuring *reliability*

[2] See point 2.8

we usually only measure defects found and defects fixed; if the goal is to fully measure *reliability*, we need to address prevention.

It comprises three different sub-characteristics:

Fault tolerance

ISO 9126 defines *Fault Tolerance* as the attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface.

In the company web page context, *Fault Tolerance* will not be directly addressed by the web site since many features of browser technology are inherently fault tolerant: If a link fails, it is possible to go "back", and if a graphic fails then it is not displayed.

Maturity

Maturity is defined by ISO 9126 as those attributes of software that bear on the frequency of failure by faults in the software.

In the company web page context, *Maturity* is not a major issue. In the context of web sites, this depends on the servers, browsers, and entire web. Apart from reducing the size of pages, there is little that can be done to help.

Recoverability

According to ISO 9126, *Recoverability* comprises those attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort required for it.

In the company web page context, *Recoverability* is handled by browser technology, which allows backtracking from failed links.

5.3.5.4 Usability

Usability is the set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stayed or implied set of users. Also referred to as “easy of use” and “human engineering”, *usability* addresses the myriad factors that constitute “user-friendliness”. It measures the effort required to prepare input for, operate and interpret the output of the system, as well as how easy it is for new or infrequent users to learn how to use the product.

Usability comprises three different sub-characteristics:

Understandability

ISO 9126 defines *Understandability* as those attributes of software that bear on the users' effort for recognizing the logical concept and its applicability.

In the company web page context, *Understandability* is of major importance. It depends on the extent to which the web page relates to the intuition of the user. Addressed topics will be similar to “No pre-requisite knowledge must be required to comprehend the contents and the structure of this site” or “The significance of the messages (under management science and multimedia) must be clear to any practitioner of the fields”.

Learnability

Learnability is defined by ISO 9126 as those attributes of software that bear on the users' effort for learning its application (for example, operation control, input, and output).

In the company web page context, *Learnability* is also of major importance. The learning process can be speeded up by means of applying to the user’s imagination. It will cover things such as “The message of the site must be apparent on a single reading (although if the message is important to the user/participant, then re-reading is inevitable given the compact nature of web-writing)”, or “The time to learn how to use the web site effectively should be as little as possible, for a semi-mature web user”.

Operability

ISO 9126 defines *Operability* as those attributes of software that bear on the users' effort for operation and operation control.

In the company web page context, *Operability* is being influenced by the “individualizing phenomena”: the greater the user specifics, the more easier the user will find it to use ("user-fit"). It will address topics such as “Having become used to its structure, operability must not interfere with the user's ability to swap between topics, or identify their current position”.

5.3.5.5 Efficiency

Efficiency is the set of attributes that bear on the relationship between the level of performance of the software and the amount of resources, under stayed conditions. It is a measure of how well the system utilizes processor capacity, disk space, memory or communication bandwidth. If a system is consuming all available resources, users will notice degraded performance, a visible indication of inefficiency. Poor performance can simply be an irritant to the user who is waiting on the program to start up, or it can represent a serious risk to safety, as when a real time control system is overloaded.

Efficiency comprises two sub-characteristics:

Time Behaviour

According to ISO 9126 *Time Behaviour* is comprised of attributes of software that bear on response and processing times and on throughput rates in performing its function.

In the company web page context, *Time Behaviour* will be presented by issues such as “It must be possible to access the site using a mobile phone, i.e. 9600 BPS maximum and sometimes worse. Use no video or animation to increase performance”.

Resource Behaviour

Resource Behaviour is defined by ISO 9126 as those attributes of software that bear on the amount of resources used and the duration of such use in performing its function.

In the company web page context, *Resource Behaviour* will be mostly related to HD space: “The site must fit within XMb. This corresponds to the allowance of server accounts and also limits the cache space requirements on user’s machines.”

5.3.5.6 Maintainability

Maintainability is the set of attributes that bear on the effort needed to make specified modifications. It indicates how easy is to correct a defect or make a change in the software. But if software does not break and it does not wear out, why *maintain* it? Because of three main reasons:

Software is rarely perfect when delivered

Software’s operating environment changes

Software’s specifications changes

According to these three points, a better term to describe *Software Maintainability* would be *Long Term (Extended) Development*, and therefore could be identified as the most expensive component of Software Development.

Maintainability is composed by four sub-characteristics:

Analyzability

According to ISO 9126, *Analyzability* is defined as those attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

In the company web page context, *Analyzability* will be related to the use of web design tools, with full tracing of links and used files, instead of just pure HTML code.

Changeability

ISO 9126 defines *Changeability* as those attributes of software that bear on the effort needed for modification, fault removal or for environmental change.

In the company web page context, *Changeability* will be also related to the use of web design tools, with full tracing of links and used files, instead of just pure HTML code.

Stability

Stability is defined by ISO 9126 as those attributes of software that bear on the risk of unexpected effect of modifications.

In the company web page context, *Stability* will cover topics like “Use only features that work on Netscape 3 and Internet Explorer 3.”

Testability

Testability is defined by ISO 9126 as those attributes of software that bear on the effort needed for validating the modified software.

In the company web page context, *Testability* would cover the use of automated tools to check whether the Hyperlinks to other sites are still valid.

5.3.5.7 Portability

Portability is the set of attributes that bear on the ability of software to be transferred from one environment. It measures the effort required to migrate a piece of software from one operating environment to another. Portability goals should state the portions of the system that must be able to migrate to other environments and identify those target environments.

Most portability problems are not pure language issues. *Portability* has to involve hardware (byte order, device I/O) and software (utility libraries, operating systems, run-time libraries, character sets).

According to ISO, *Portability* has four different sub-characteristics:

Adaptability

In ISO 9126 Adaptability is defined as those attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered.

In the company web page context, *Adaptability* is not an important issue. Compatibility with Netscape 4 and Explorer 4 is not important when there is an increasing number of mobile terminals that are likely to stay with version 3 browsers for the next 2-3 years.

Conformance

ISO 9126 defines *Conformance* as those attributes of software that make the software adhere to standards or conventions relating to portability.

Installability

Installability is defined by ISO 9126 as those attributes of software that bear on the effort needed to install the software in a specified environment.

In the company web page context, *Installability* is not addressed at all as no installation is required.

Replaceability

Replaceability is defined by ISO 9126 as those Attributes of software that bear on the opportunity and effort of using it in the place of specified other software in the environment of that software. *Replaceability* is used in place of compatibility in order to avoid possible ambiguity with interoperability.

In the company web page context, *Replaceability* is not a major issue as the only external interfaces to be kept (links to other pages) are very easy to handle in new pages.

5.3.5.8 Quality in Use

Quality in use is the user's view of quality. It is closely related to *Usability*, but it is actually a broader term than the former since it could be defined as the combination of product attributes that provide the greatest satisfaction to a specific user. It can be seen as the "external" software product quality attributes in contrast to the "internal" ones (*Functionality, Reliability, Portability…).*

It is much more qualitative than quantitative as it tries to incorporate human factors to the software quality identification process, and because of that extremely difficult to be measured. **We will not focus on it**.

It has four sub-characteristics:

Effectiveness

ISO 9126 defines *Effectiveness* as the capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specific context of use

Productivity

Productivity is defined by ISO 9126 as the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specific context of use.

Safety

Safety is defined by ISO 9126 as the capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specific context of use.

Satisfaction

According to ISO 9126, *Satisfaction* is the capability of the software product to satisfy users in a specific context of use.

5.3.6 NFR Analysis and Decomposition

In this chapter we go through the ISO 9126 quality attribute list showing particular decomposition patterns for each quality attribute. Note that this experience-based decomposition approaches are as general as they can be, and are not tailored for any particular framework. They can be used as starting point or as an example's source for the Empress elicitation, documentation and analysis process.

5.3.6.1 Functionality

This characteristic relates to the achievement of the basic purpose for which the software is being developed. It is concerned with what the software does to fulfil needs, whereas the other characteristics are mainly concerned with when and how it fulfil needs.

In Figure 5-41 a first level decomposition of functionality can be seen. Four of the subcharacteristics ("Accuracy", "Interoperability", "Security" and "Suitability") are taken from the ISO 9126 specification and a fifth subcharacteristic ("Backwards compatibility") has been added. There are more detailed explanations about them in the chapters 5.3.6.1.1 to 5.3.6.1.5.

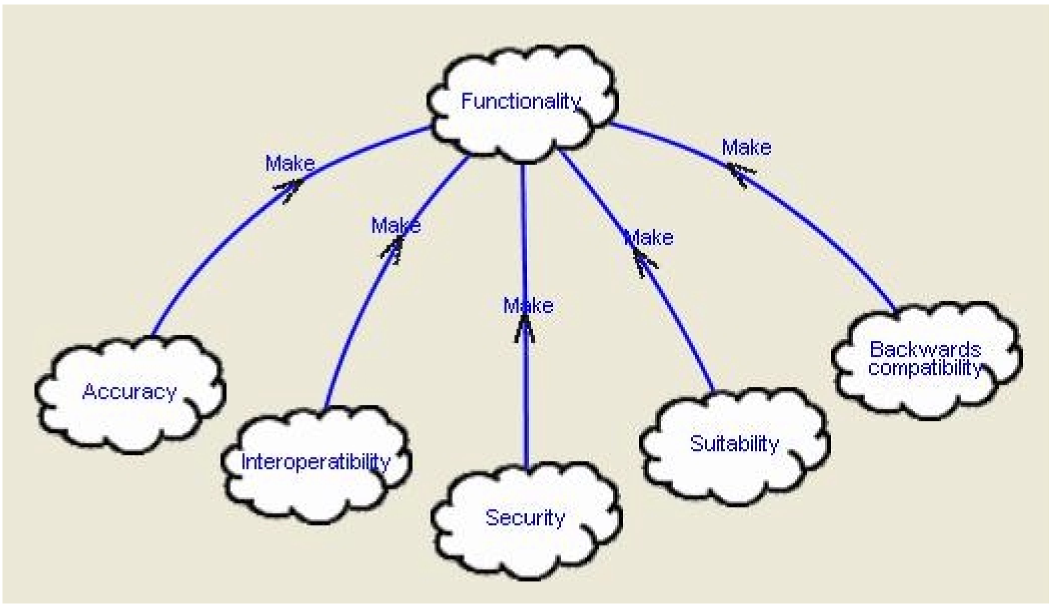


Figure 5-41: Functionality decomposition

The user will be able to verify the functionality of the system through the quality in use of the system.

In the McCall's model [McC94] this characteristic is called correctness ("Extent to which a program satisfies its specifications and fulfils the user's mission objectives").

5.3.6.1.1 Accuracy

An accurate system gives the right information at the right time. A decomposition of this subcharacteristic can be seen in the following figure (Figure 5-42).

"Value accuracy" means that the values used in the calculations and returned are accurate. This kind of accuracy is especially important if some calculations with decimal numbers or divisions have to be done. For example, a first idea for doing the sum of many amounts of cents coins (x1, x2, x3…) in order to change them into currency units (€, $...) without cents in a system that only supports integers could be x1/100+x2/100+x3/100+… but this calculation is quite inaccurate and a much better accuracy would be achieved in the same system if the calculation was done this way: (x1+x2+x3+…)/100

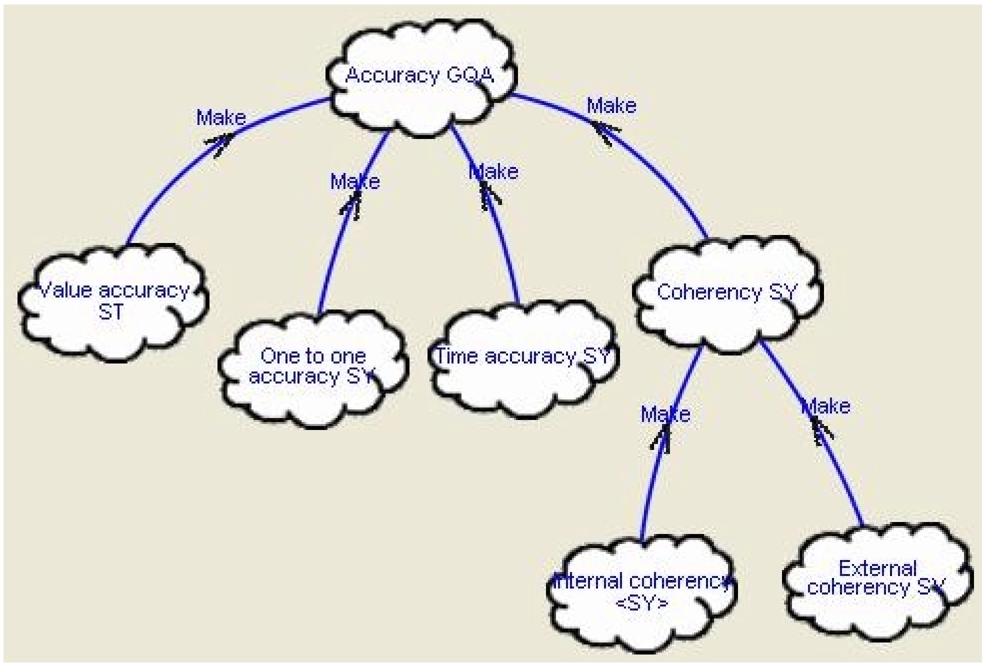


Figure 5-42: Accuracy decomposition

There is “One to one accuracy” when the information is associated with the entities with which it should be associated and only with them. The freshness of the information improves the one to one accuracy. For example, an internet browser with cache memory could show inaccurate information if the cache memory isn’t updated correctly.

“Time accuracy” is the correspondence between system time and real time. A perfect clock which gives always the exact time doesn’t exist, but the question is if the clock is accurate enough for the system. A system only has to be accurate enough for its tasks. For example, in a Microsoft Windows desktop operating system a time accuracy of seconds will be enough. A time accuracy of milliseconds can be demanded to a real time system.

The “Coherency” of the system is the validity of the relationships among values in the system, “Internal coherency” if the related values are inside of the system or “External coherency” if internal values are related to external values. The internal coherency is related to the validity of relationships among values inside of the system. For example, if one value of the system is the measurement of one sensor and can be also calculated from the measurement of other sensors, a system with internal coherency would use always the value taken with one of the methods or one obtained from the application of a certain formula (but always the same, for example, an average value). If the system is accurate enough (not only the software, but also the values from the sensors), there won’t be differences between the values obtained with both methods and the system will have value accuracy, but it won’t have internal coherency. One system has external coherency when there is a correspondence between external and internal values. The values have to be coherent in the working environment of the system and to correspond not with the introduced data but with the actual data. A user failure in the introduction of data could be a source of external coherency because of non-correspondence with the data which should be actually introduced and even incoherent values in the working environment (for example, a value of 120°C for the temperature of a boiling pot at atmosphere pressure). Coherency is used in place of consistency in order to avoid ambiguity with the attribute of Learnability that has already that name.

The accuracy was already seen as a subcharacteristic of a software system in previous quality models (Boehm's [Boe78] and McCall [McC94]). In both models the accuracy is a subcharacteristic of the reliability of the system.

This subcharacteristic has some influence in other attributes of the software; it will especially deteriorate its efficiency. Many artefacts added to improve the accuracy of the system will usually get the efficiency of the system worse. For example, some "Redundancy" in the system (for example, data that are calculated using two different ways) can improve the accuracy, but will also get the efficiency worse. The accuracy can also have some negative effects in the operability of the system; for example, if some mechanisms have to be added to check the accuracy of the information introduced by the user (asking for confirmation, asking for a double introduction on the same value…).

5.3.6.1.2 Backwards compatibility

This subcharacteristic is the ability of a system to run instead of an older version of the same system. It's very important to notice that this attribute is only valid for a new version of a software system substituting an older version of the same system. Obviously, the first version of a certain system won't ever have backwards compatibility. The fewer the necessary modifications in the new version are so that it can always replace any order version, the better the backwards compatibility of the system will be. A system will have full backwards compatibility not only if it has "Interface backwards compatibility", but also if it has "Requirements backwards compatibility". That decomposition can be seen in the following figure (Figure 5-43).

A system which has "Interface backwards compatibility" has the interface of the previous version of the system or more extended. For example, a new version of a module that has a function that has to be called with more parameters than in the older versions won't have interface backwards compatibility. In order to maintain the interface backwards compatibility of the system, both possibilities should exist: calling the function only with the parameters of the older versions and calling the function with all the parameters of the new version. If there are many changes between the interface of the new version and the old interface that want to be maintained, the interfacing part of the system will become much more complicated. These complications don't add functionality to the system and they do load the system, that's why in some cases the interfacing backwards compatibility won't be desirable.

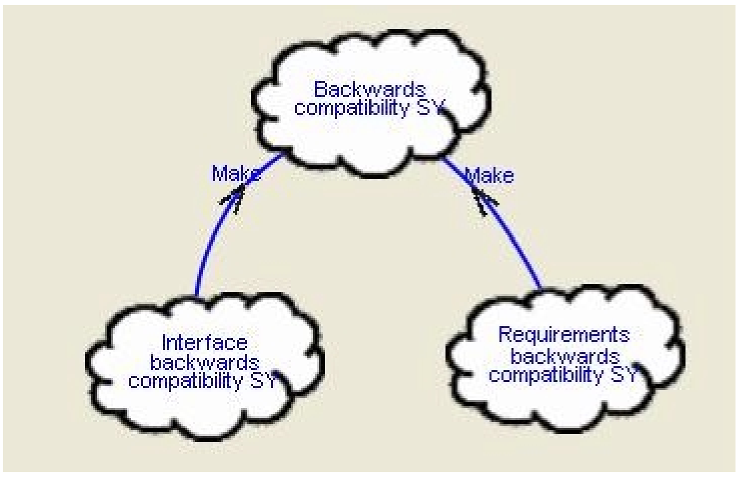

Backwards compatibility SY
Make
Make
Interface backwards compatibility SY
Requirements backwards compatibility SY

Figure 5-43: Backwards compatibility decomposition

The system will have “Requirements backwards compatibility” if all the hardware and software requirements are either the same or less restrictive. The fact that the system works perfectly in a working environment where an older version worked doesn’t mean that it has requirements backwards compatibility. Let’s see an example, there are two versions of one system and the only difference between them is that the new version requires 60Mb of RAM and the older one only 50Mb, but the old version was installed in a working environment with 64Mb of RAM. In this case the new system can substitute the old one without problems, but the new one has not requirements backwards compatibility, because in working environments with less than 60Mb of RAM the new version could not substitute the old one. Sometimes it isn’t worth maintaining the old requirements (for example, if the new version of the software will be mainly installed in a new working environment) in order not to restrict the system improvements.

5.3.6.1.3 Interoperability

The interoperability of a system can be defined as the effort required to couple or interface the system with other systems (therefore this subcharacteristic is sometimes also named interface facility) or as the ability of the system to interact with other systems, exchanging data, services… A decomposition of the subcharacteristic Interoperability can be seen in Figure 5-44.

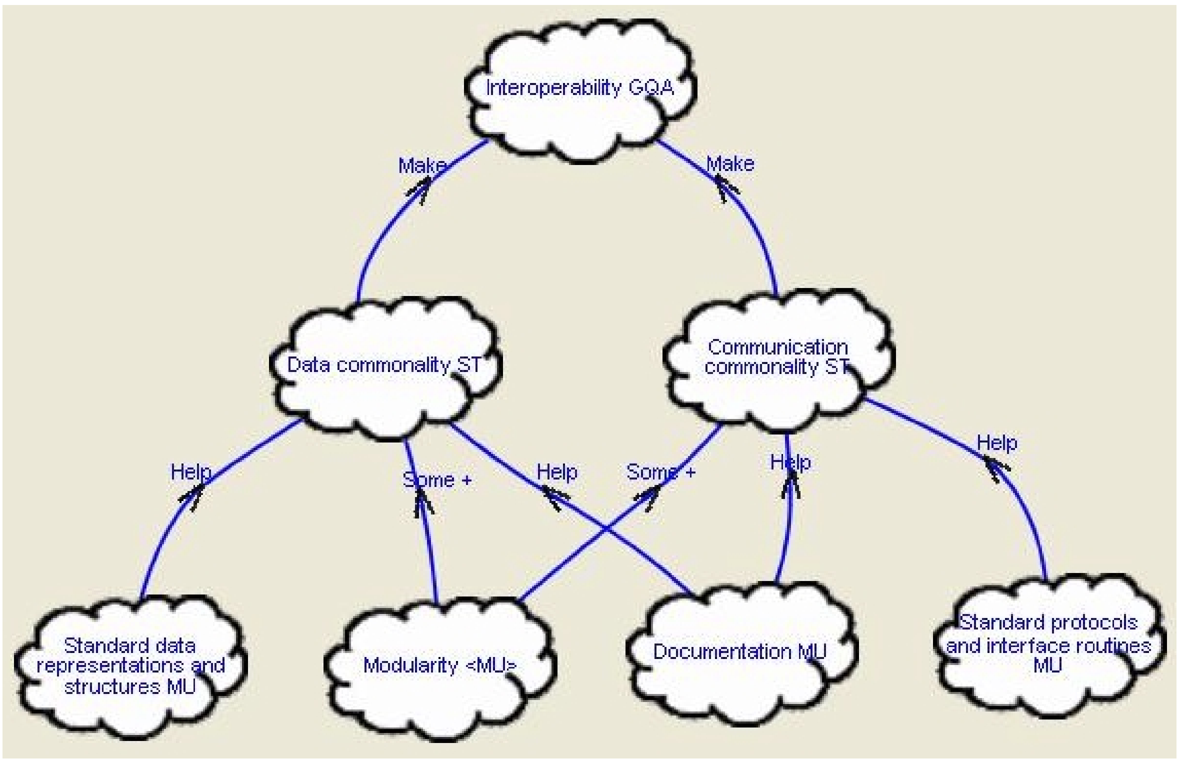

Interoperability GQA
Make
Make
Data commonality ST
Communication commonality ST
Help
Some +
Help
Some +
Help
Help
Standard data representations and structures MU
Modularity <MU>
Documentation MU
Standard protocols and interface routines MU

Figure 5-44: Interoperability decomposition

In order to have this coupling or interaction, the systems have to fulfil two conditions:

“Communication commonality”: The system and the other systems have to be able to communicate. Communication has to be understood as the interfacing of the systems (APIs) and the use of communication protocols. For example, if they communicate via a local area network, they have to use the same network protocol. The possibility of having communication commonality between the system and the other systems is increased if the system supports “Standard protocols and interface routines”, because the probability of trying to communicate with a system that supports a standard protocol or interface routine is bigger than that of trying to communicate with a system which doesn’t. For example, the POSIX interfaces are a good example of standard interface routines.

“Data commonality”: The system and the other systems have to be able to interpret the data that they share in order to extract the information. Let’s put it in other words: they must have a common way of interpreting the data they’re sharing. The possibility of having some data commonality is increased if “Standard data representations and structures” are used for storing the information. For example, if the system stores an image in a JPEG format (a standard data representation for images), the probability of having a system that supports JPEG trying to get the image is much higher than if the image is stored in a proprietary format.

One thing that has some good influence in both kinds of interactions is the “Modularity” of the system. Modularity can be defined as that attribute of the software that provides a structure of highly independent modules with each serving a particular function.

The “Documentation” of the protocols and data structures used in the part of the system that will interact with other systems is very important in order to allow a proper interaction. It’s especially important if some no standard protocols or data structures have been used.

The interoperability of the system is a very important attribute of the systems that could be integrated in bigger systems. For example, a library that will be installed in an operating system.

The use of APIs, plug-in protocols and those very used interfacing artefacts improves the interoperability of the system.

This subcharacteristic is also present in the McCall's model [McC94] and with the same name and definition.

It's important emphasizing that the interoperability of a system is a very positive factor for the portability and testability of the system. The coupling capabilities of the system with the new working environment have quite a lot of influence in its portability. It's very important implementing some self testing mechanisms in the system, but many times some external systems are necessary for the tests and in those cases the interfacing capabilities of the system are very important.

5.3.6.1.4 Security

This subcharacteristic qualifies the runtime behaviour of the system. It can be decomposed as it can be seen in the following figure (Figure 5-45):

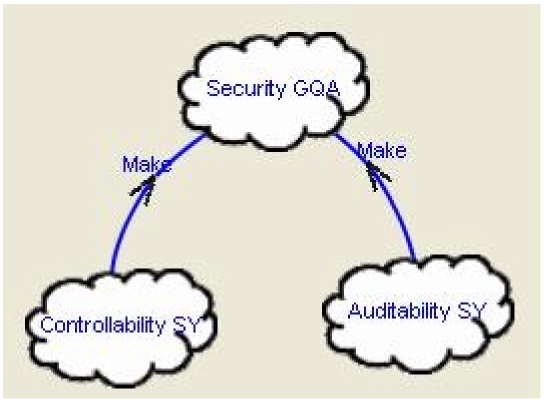


Figure 5-45: Security decomposition

The attribute that applies to a system that blocks unauthorized accesses is the "Controllability". The control of the accesses to the system (letting the authorized and blocking the unauthorized) is the most important aspect of a system that has to be secure. The system has to give permissions to consult or update its information when necessary and to guarantee the data confidentiality. This attribute is called integrity in the McCall's quality model [McC94]: "extent to which access to software or data by unauthorized persons can be controlled". There are many means to control accesses:

If the data can be somehow accessed without using the system (for example, due to network communication), the use of "Data encryption" will be the only way to avoid unauthorized accesses to the information so that those accesses can be controlled. For example, the encryption of all the data flowing in a network or stored in a hard disk will improve the security of the system responsible for those data.

The "Identification" of external user or systems. Who is trying to access the system? But the system can not do anything only identifying a user.

It also has to check which "Authorizations" the user has. This way the system knows which users have permissions to access which data or functions.

The “Authentication” of the identity is fundamental in order to avoid that a user tries to enter as if he or she was another user in order to get more permissions. The normal way to do it is the use of passwords.

“Auditability” is the attribute of a system that has some auditing mechanism which records users’ access to the system and data. This attribute is relevant if there has been an unauthorized access, because it helps to minimize the influence of the break into or, when it’s already too late, to repair the security holes that made it possible. The implementation of “Intrusion detection” mechanisms is necessary for the auditability of unauthorized accesses.

This attribute will usually have a negative effect in the efficiency of the system because the artefacts implemented to improve the security of the system will usually load the system. The improvements in the controllability of the system and the data encryption will usually make easier the exchange of data and services, thus impeding the interoperability of the system.

5.3.6.1.5 Suitability

This subcharacteristic could be divided in three different attributes (as it can be seen in Figure 5-46).

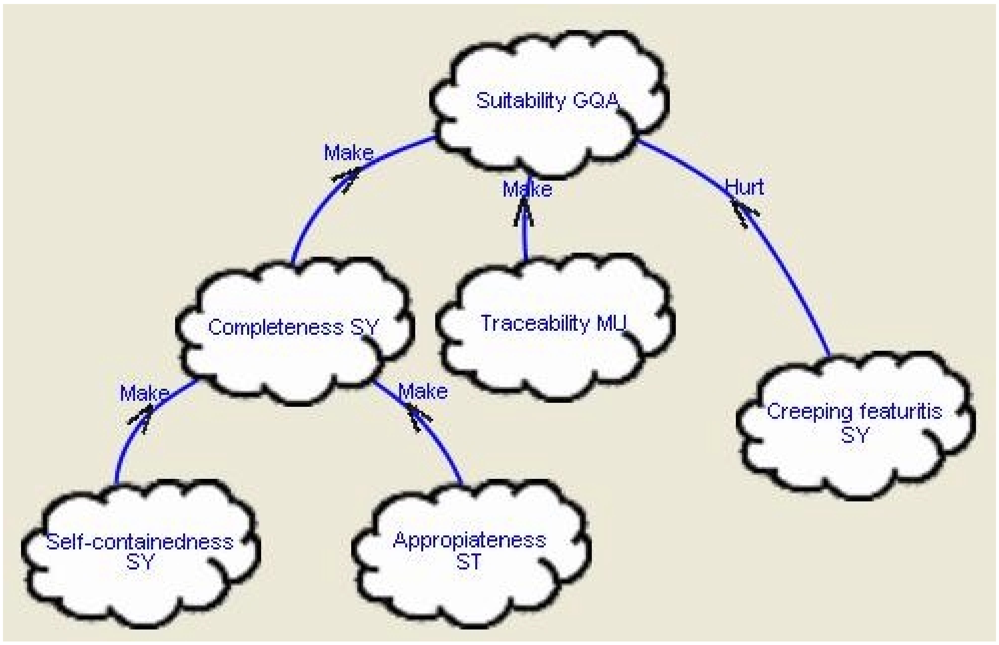


Figure 5-46: Suitability decomposition

The more required features or functions implemented by the system, the better the “Completeness” is. Of course, a well designed system should cover all the required features and functions. Although at first sight this could be considered the most important attribute of a system, sometimes not implementing a feature is better than implementing it wrong, especially those in which the human being security can be involved.

The “Completeness” can be decomposed in “Self-containedness” and “Appropriateness”. If the system is not self-contained some additional precautions have to be taken in order to assure its completeness. For example, if the system is an application for Microsoft Windows, some mechanisms should be implemented to assure that the necessary DLLs are present and to install them, in case they aren't present. Sometimes it's impossible (it is necessary using some external libraries) and some other times a self-contained system is not desirable because other characteristics of the system (for example, modularity) are more important for the system than this one. The system not only has to offer all the necessary features, but they also have to be appropriate for the specified task. The "Appropriateness" of the features implemented is very important, the only presence of a feature doesn't do it suitable for its task. For example, if the system implements some kind of drawing facility for block diagrams, a facility in which one element can't be repositioned once placed is not appropriated (even if it isn't specified as a requirement).

Does the system offer too many features? If there some features that aren't required and will never be used are implemented in the system, the system suffers from "Creeping featuritis". This excess of features will usually have negative effects in other requirements. For example, the understandability of the system will get worse.

In order to be able to check the suitability of the system, it's quite interesting to have a connection thread from the requirements to the implementation. The "Traceability" is the attribute of a system that has those connection threads. A traceable system will be probably complete, because when creating this connection between features and requirements a double checking will be probably done: requirement → feature (which feature or features implement a certain requirement?) and feature → requirement (does a requirement motivated a certain feature or features? which requirement?).

This equivalent subcharacteristic in the Boehm's model [Boe78] is called completeness: "all its parts are present and each of its parts is fully developed". In the McCall's model [McC94] the equivalent factor is called correctness: "extent to which a program satisfies its specifications and fulfils the user's mission objectives".

5.3.6.2 Reliability

Figure 5-47 shows the basic decomposition of system reliability. The first three softgoals ("Fault Tolerance", "Recoverability" and "Maturity") are results of the ISO 9126 specification and they are covered in more detail on chapters 5.3.6.2.1, 5.3.6.2.2 and 5.3.6.2.3.

The fourth one ("Availability") is an extra softgoal to reflect (in a better and more direct way) whether the system is up (available, working or above a given quality-of-service level) or down (not available, not working or below the quality-of-service level). For non-repairable systems, availability and reliability are equal. For repairable systems they are not, and as a general rule it can be said that $0 \leq R(t) \leq A(t) \leq 1$. It will be covered on chapter 5.3.6.2.4.

Finally, the fifth softgoal ("Safety") is used to mean the extent to which a system is free from system hazards, and it also covers the prevention and/or reduction of accidents throughout the life cycle of the system. It will be addressed on chapter 5.3.6.2.5.

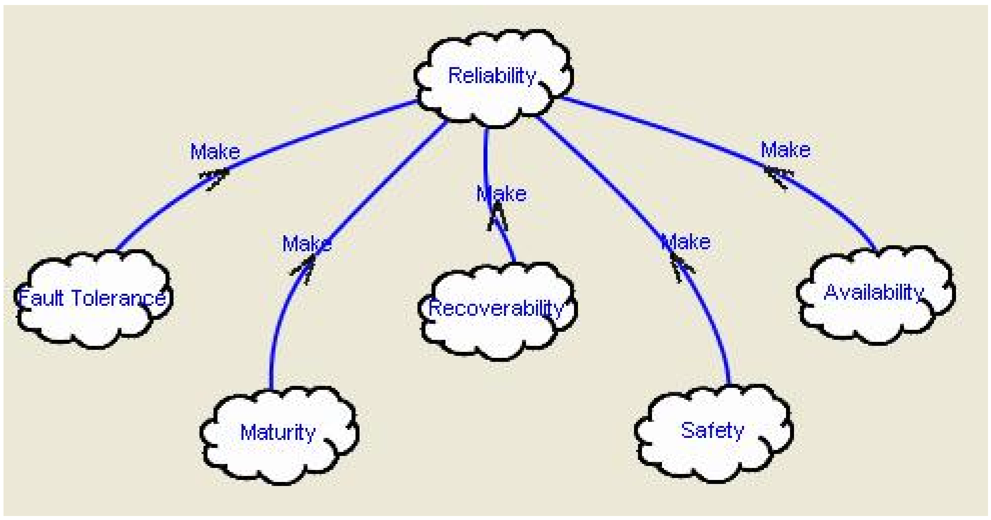


Figure 5-47: Reliability decomposition

Since the coupling between SW and HW is so high in embedded systems, it is tempting to draw an analogy between software reliability and hardware reliability. This approach could be totally wrong as software and hardware have basic differences that make them different in failures mechanisms. A partial list of the distinct characteristics of software compared to hardware is listed below:

Failure cause: Software defects are mainly design defects; hardware defects are mostly physical faults.

Wear-out: Software does not have energy related wear-out phase, and HW definitely does: Time makes HW reliability go down as it wears out.

Repairable system concept: Periodic restarts can help fix software problems.

Time dependency and life cycle: Software reliability is not a function of operational time.

Environmental factors: Do not affect Software reliability, except it might affect program inputs.

Reliability prediction: Software reliability can not be predicted from any physical basis, since it depends completely on human factors in design.

Redundancy: Cannot improve Software reliability if identical software components are used.

Interfaces: Software interfaces are purely conceptual other than visual.

Failure rate motivators: Usually not predictable from analyses of separate statements.

Built with standard components: Well-understood and extensively-tested standard parts will help improve maintainability and reliability. But in software industry, we have not observed this trend. Code reuse has been around for some time, but to a very limited extent. Strictly speaking there are no standard parts for software, except some standardized logic structures.

This different is not directly addressed in the decomposition diagrams shown here. We will explain common approaches that can be used to analyze HW and SW reliability, but it is a reader’s task to differentiate the two fields when further refining the decomposition trees.

Another important remark is the difference between *fault*, *error, failure*, *hazard* and *accident*. A **fault** is a deviation of the behaviour of a computer system from the authoritative specification of its behaviour. A hardware fault is a physical change in hardware that causes the computer system to change its behaviour in an undesirable way. A software fault is a mistake (also called a bug) in the code. A user fault consists of a mistake by a person in carrying out some procedure. An environmental fault is a deviation from expected behaviour of the world outside the computer system; electric power interruption is an example. “Fault Tolerance” is the softgoal that deals with faults.

An **error** is an incorrect state of hardware, software, or data resulting from a fault. An error is, therefore, that part of the computer system state that is liable to lead to failure. Upon occurrence, a fault creates a latent error, which becomes effective when it is activated, leading to a failure. If never activated, the latent error never becomes effective and no failure occurs. “Maturity” is the softgoal that addresses errors.

A **failure** is the external manifestation of an error. That is, a failure is the external effect of the error, as seen by a (human or physical device) user, or by another program. “Recoverability” is the softgoal that covers failures.

Some examples may clarify the differences among the first three terms. A fault may occur in a circuit (a wire breaks) causing a bit in memory to always be a 1 (an error, since memory is part of the state) resulting in a failed calculation. A programmer's mistake is a fault; the consequence is a latent error in the written software (erroneous instruction). Upon activation of the module where the error resides, the error becomes effective. If this effective error causes a divide by zero, a failure occurs and the program aborts. A maintenance or operating manual writer's mistake is a fault; the consequence is an error in the corresponding manual, which will remain latent as long as the directives are not acted upon.

An **accident** is an unplanned event or series of events that result in death, injury, illness, or environmental damage. A **system hazard** is an application system condition that is a prerequisite to an accident. That is, the system states can be divided into two sets. No state in the first set (of non-hazardous states) can directly cause an accident, while accidents may result from any state in the second set (of hazardous states). Note that a system can be in a hazardous state without an accident occurring—it is the potential for causing an accident that creates the hazard, not necessarily the actuality.

5.3.6.2.1 Fault Tolerance

Figure 5-48 shows the decomposition pattern for Fault Tolerance. Since “Fault Tolerance” deals with faults, the basic analysis process is based on the types of faults. Further refinement can be easily done applying a HW/SW sub-system decomposition approach, and finding out how the system has to behave on the resulting part regarding a particular fault type.

“Input Data Fault Tolerance” and “Output Data Fault Tolerance” address mistakes on the system’s input and output processes (buffer overrun, for example). “Interface Fault Tolerance” covers the internal communication interfaces and throughput channels, including dynamic changes of intra-subsystem interfaces. “Data Definition Fault Tolerance” deals with logical definition and description of data (negative values for an unsigned integer).

A “permanent state fault” is a fault in state data that is stored on non-volatile storage media (erroneous value recorded on a log-file). On the other hand, a “temporary state fault” happens when it is recorded on volatile storage media (such as main memory).

“Task faults” are related to task synchronization. Deadlocks, race conditions and zombie threads are typical examples.

An “operator fault” is a mistake by the operator. Any of the three following types are possible. A design fault occurs if the instructions provided to the operator are incorrect; this is sometimes called a procedure fault. An operational fault would occur if the instructions are correct, but the operator misunderstands and doesn't follow them. A transient fault would occur if the operator is attempting to follow the instructions, but makes an unintended mistake. Hitting the wrong key on a keyboard is an example. (One goal of display screen design is to reduce the probability of transient operator errors).

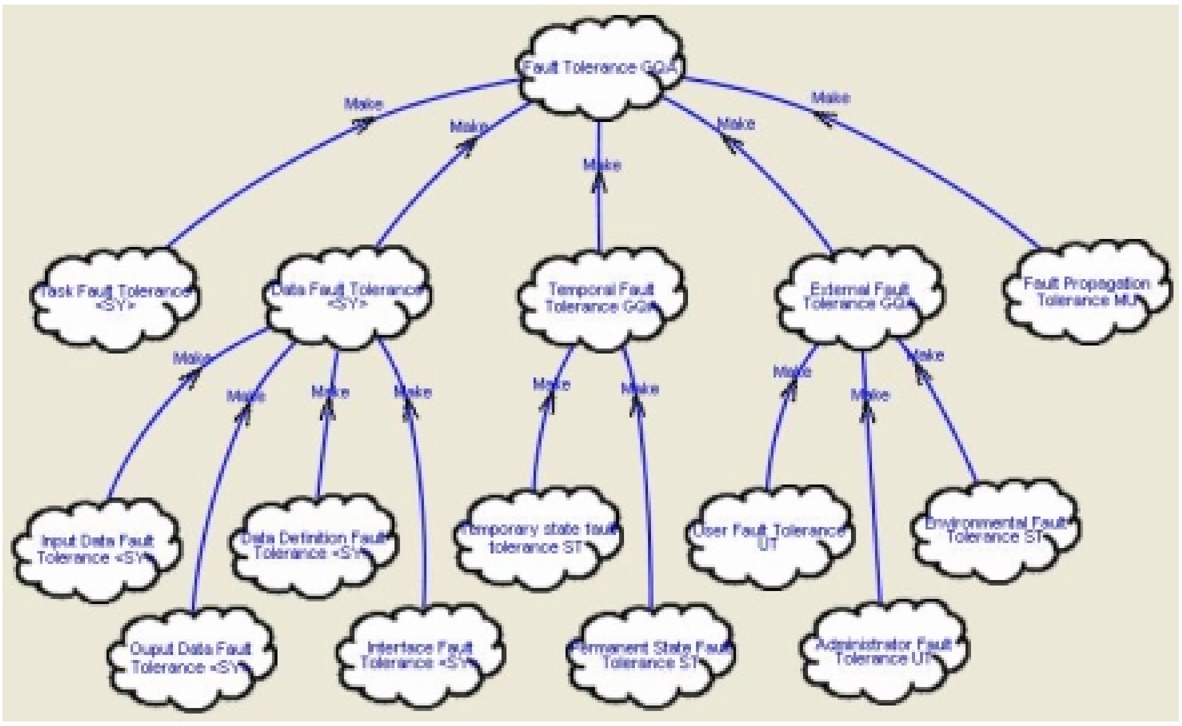

Fault Tolerance GQA
Make
Make
Make
Make
Make
Task Fault Tolerance <SY>
Data Fault Tolerance <SY>
Temporal Fault Tolerance GQA
External Fault Tolerance GQA
Fault Propagation Tolerance MU
Make
Make
Make
Make
Make
Make
Make
Make
Make
Input Data Fault Tolerance <SY>
Data Definition Fault Tolerance <SY>
Temporary state fault tolerance ST
User Fault Tolerance UT
Environmental Fault Tolerance ST
Ouput Data Fault Tolerance <SY>
Interface Fault Tolerance <SY>
Permanent State Fault Tolerance ST
Administrator Fault Tolerance UT

Figure 5-48: Fault Tolerance decomposition

A “user fault” differs from an operator fault only because of the different type of person involved; operators (or administrators) and users can be expected to have different fault rates.

An “environmental fault” is a fault that occurs outside the boundary of the computer system, but that affects the system. Any of the three types is possible. Failure to provide an uninterruptible power supply (UPS) would be a design fault, while failure of the UPS would be an operational fault. A voltage spike on a power line is an example of an environmentally induced transient fault.

Finally, most reliability analysis and modelling assume that each fault causes at most a single failure. That is, failures are statistically independent. This is not always true. A common-mode failure occurs when multiple components of a computer system fail due to a single fault. If common mode failures do occur, an analysis that assumes that they do not will be excessively optimistic. The “Fault Propagation Tolerance” softgoal addresses such a scenario.

5.3.6.2.2 Recoverability

Figure 5-49 depicts a basic decomposition pattern for “Recoverability”. Since this softgoal deals with failures, the analysis process is based on the types of failures, and when “Complete failure” is written, it means “Complete failure recoverability”. Further refinement can be easily done applying a HW/SW sub-system decomposition approach, and finding out how the system has to behave on the resulting part regarding a particular failure type.

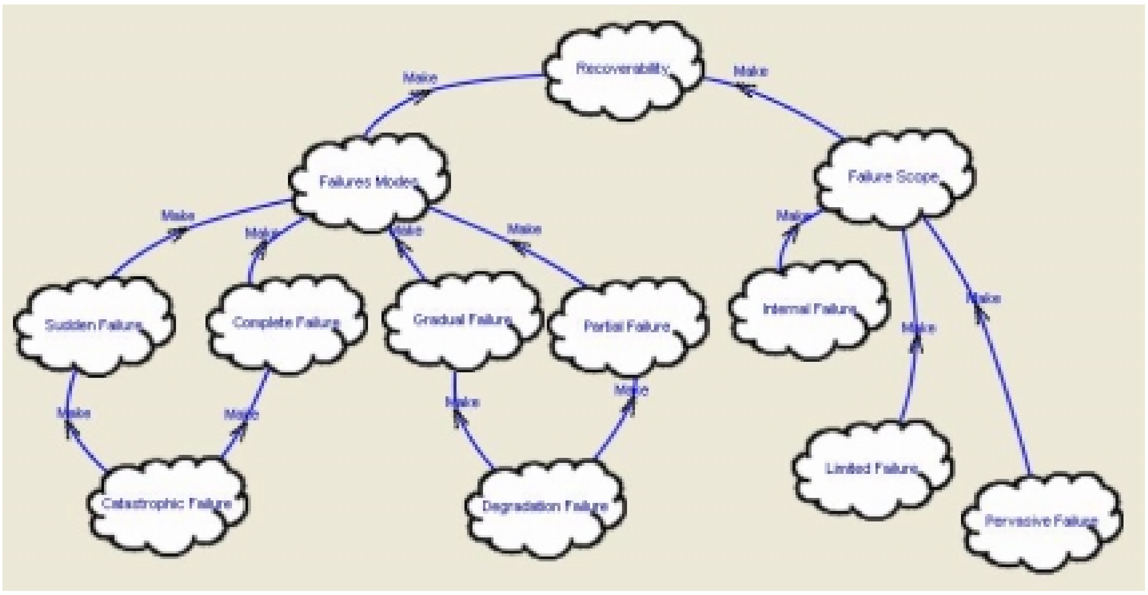


Figure 5-49: Recoverability decomposition

Different failure modes can have different effects on an embedded system.

A “sudden failure” is a failure that could not be anticipated by prior examination. That is, the failure is unexpected. A “gradual failure” is a failure that could be anticipated by prior examination. That is, the system goes into a period of degraded operation before the failure actually occurs. A “partial failure” is a failure resulting in deviations in characteristics beyond specified limits but not such as to cause complete lack of the required function. A “complete failure” is a failure resulting in deviations in characteristics beyond specified limits such as to cause complete lack of the required function. The limits referred to in this category are special limits specified for this purpose. Finally, as it is shown in the picture, a “catastrophic failure” is a failure that is both sudden and complete and a degradation failure is a failure that is both gradual and partial.

According to another division criterion, failures can be assigned to one of three classes, depending on the scope of their effects. A failure is “internal” if it can be adequately handled by the device or process in which the failure is detected. A failure is “limited” if it is not internal, but if the effects are limited to that device or process. A failure is “pervasive” if it results in failures of other devices or processes.

5.3.6.2.3 Maturity

Figure 5-50 depicts a basic decomposition pattern for “Maturity”. Once again, further refinement can be easily done applying a HW/SW sub-system decomposition approach.

The “State of the Art” softgoal analyzes the maturity of the knowledge and techniques used for the system development. It is further split into “Standards” (How mature and accepted are the standards that the system follows?, standards are usually mature and their use will usually improve the maturity of a system), “Technology” (Is the technology –Bluetooth, for example- that the product is based-on mature enough?) and “Processes and Methods” (Does the methodology –extreme programming, for instance- and/or the software development process meet a minimum maturity degree to develop the system successfully?).

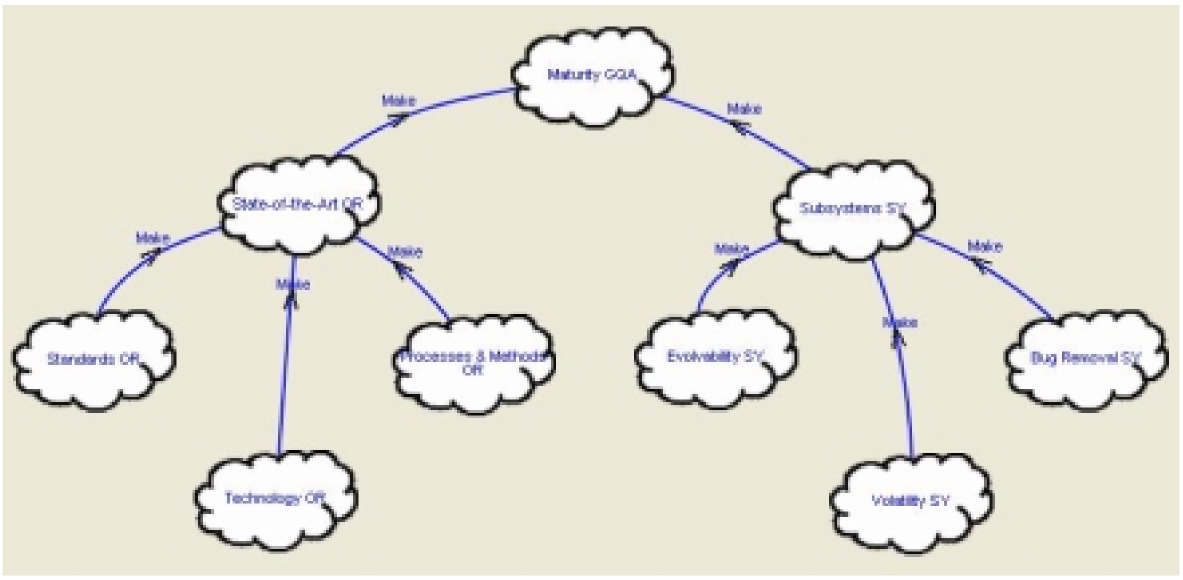


Figure 5-50: Maturity decomposition

The softgoal called “Subsystems” covers third-party software that has to be integrated into the system. Software developers do not normally have full access to this code, and because of that they have to rely on other people capability. In order to analyze this situation, three more softgoals have been included:

Bugs Removal: This softgoal evaluates the presence of bugs in a system. A maturity aspect is the number of bugs fixed in a given version of the component. Although a priori a high number of bugs fixed in a version could indicate that the new version is more stable, the authors’ experience shows that the more bugs discovered in a product, the more bugs remain to fix. Thus, the number of fixed bugs is a good measure of the bugs that still remain hidden. Note that this softgoal can also be applied on self-developed software.

Evolvability: This attribute deals with a part of the history of a system, its evolution, as a way of evaluating its maturity. The number of versions that have been marketed for a given component may provide an indication about the product maturity, and how it has evolved from its first version. This attribute may be interesting in conjunction with Volatility to indicate whether new versions are expected in a short time or not.

Volatility: This softgoal analyzes another part of the history of a system, not analyzed by the evolvability. This softgoal analyzes the average time between commercial versions. We have named it “Volatility” because it gives us an indication about the life time of a version in the market.

The relationship between Evolvability and Volatility is quite narrow and they both have to be used together to analyze the history of a system as a way to evaluate its maturity. It is not very useful knowing that there are 3 versions of a system (as a measure of its evolvability), because it is expected that a system whose versions appear approximately each year will be more mature than one whose versions appear each 3 months. It is also not very useful knowing that till now each version of a system appear each year (as a measure of its volatility), if it is unknown the number of versions released (measure of the evolvability) or the age of the system (it can be calculated measuring the evolvability as number of versions released).

5.3.6.2.4 Availability

Although in the ISO 9126 specification is written that is not included as a separate subcharacteristic (because it is a combination of maturity, fault tolerance and recoverability), it has been included in this model. It has been done so, because availability is very often considered as a quality attribute and has specific metrics. It can be refined through its dependence on maturity, fault tolerance and recoverability.

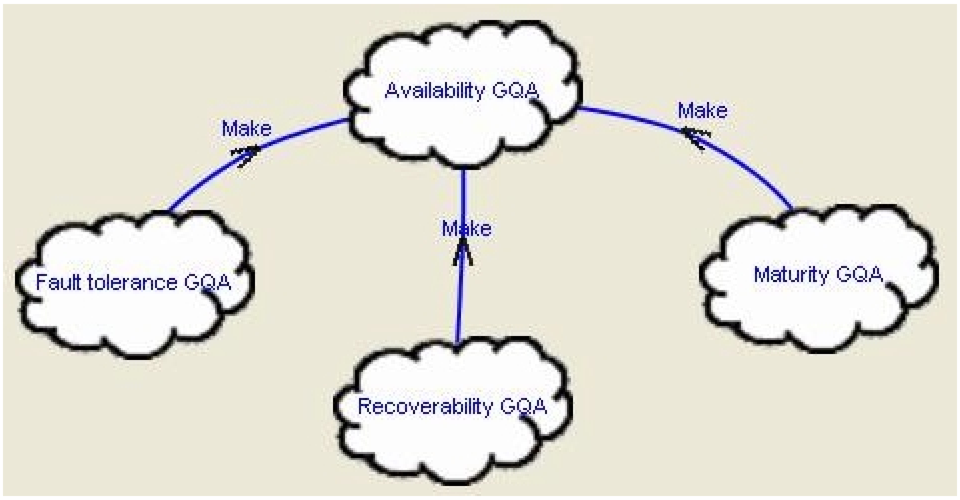


Figure 5-51: Availability decomposition

5.3.6.2.5 Safety

This subcharacteristic as availability is not present in the ISO model, but has been added because it is frequently considered a reliability subcharacteristic. Even though it can be refined as it will be seen, this quality aspect depends quite a lot on the reliability of the system (a failure in the execution of the system can take it to a hazard) and on its usability (the user should know if an action will take the system to a hazard).

Figure 19 depicts a basic decomposition pattern for “Safety”. Since this softgoal deals with hazards and accidents, the decomposition process is based on the analysis of those events. Further refinement can be easily done applying a HW/SW sub-system decomposition approach, and finding out how the different subsystems cover safety aspects.

From the hazards point of view, the first analysis step is their identification. Once all the possible hazards have been identified, the best alternative is to eliminate them as during the requirements specification or design phase, hazard elimination adds no cost. Note that there may be some hazards that cannot be eliminated from the system. The goal in this case is to minimize the occurrence of the hazard. One way to do this is to carefully control the conditions under which the system can move from a safe state to a hazardous state (“Hazard_Condiftions” softgoal). Another way is to create hazard states where the main aim is to minimize the chances that the hazard turns into an accident, and where the system prevents an accident from resulting from the hazard (“Hazardous_states” softgoal). On the weapon example, this could lead to send the arming signal as a five digit code rather than a single wire with high or low voltage. That way, a transient spike on one wire is not capable of arming the weapon. The system is comparatively safe if it is not armed. Arming the weapon puts it into a hazardous state; it may be lethal or severely damaging if triggered. But for proper functioning of a weapon, one cannot eliminate the armed state.

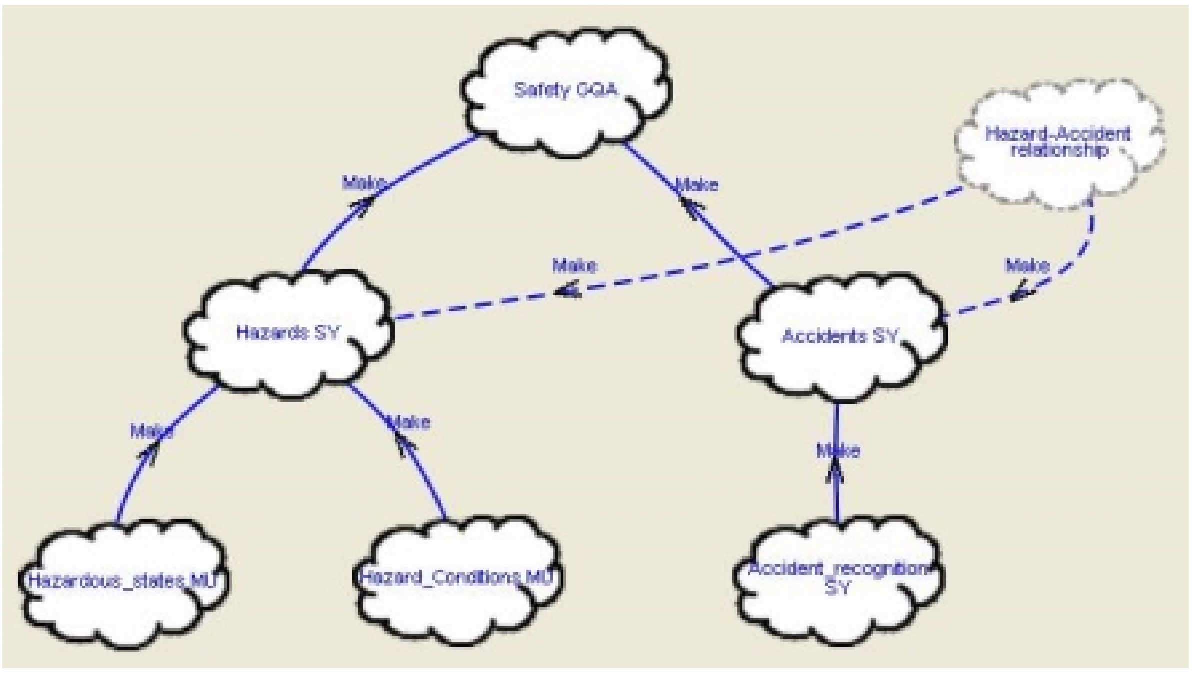


Figure 5-52: Safety decomposition

Regarding accidents, the most important things are the analysis of the relationships between accidents and hazards, and how the system behaves when an accident occurs ("Accident Recognition" softgoal). On this point, the system is said to control accidents if a hazardous state may be entered and an accident may occur, but the system will mitigate the consequences of the accident. An example is a containment shell of a nuclear reactor, designed to preclude a radiation release into the environment if an accident did occur.

A system gives warning of hazards if a failure may result in a hazardous state, but the system issues a warning that allows trained personnel to apply procedures outside the system to recover from the hazard or mitigate the accident. For example, a nuclear reactor computer protection system might notify the operator that a hazardous state has been entered, permitting the operator to "hit the panic button" and force a shutdown in such a way that the computer system is not involved.

A system is fail dangerous, or creates an uncontrolled hazard, if system failure can cause an uncontrolled accident.

Finally, two more softgoals ("Environmental Factors" and "Human Factors") are included into the decomposition pattern to address external aspects that may influence safety but cannot be directly controlled by the system itself.

5.3.6.3 Usability

This characteristic is only applicable to systems with a human computer interface. And only the software based aspects of the interface can be judged.

The evaluation of this characteristic is quite subjective and though it can be subdivided, the resulting subcharacteristics are quite subjective too. Nevertheless, the evaluation of these aspects is very important in systems with a human computer interface.

This characteristic is decomposed in three different subcharacteristics: “Learnability”, “Operability” and “Understandability”. There is a forth aspect (“Effectiveness”) that helps evaluating the usability, but will not be refined because of the total dependency from the three subcharacteristics. Effectiveness can be defined as the capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use, supposing that the functionality of the system is complete and that it is reliable. This decomposition can be seen in Figure 5-53.

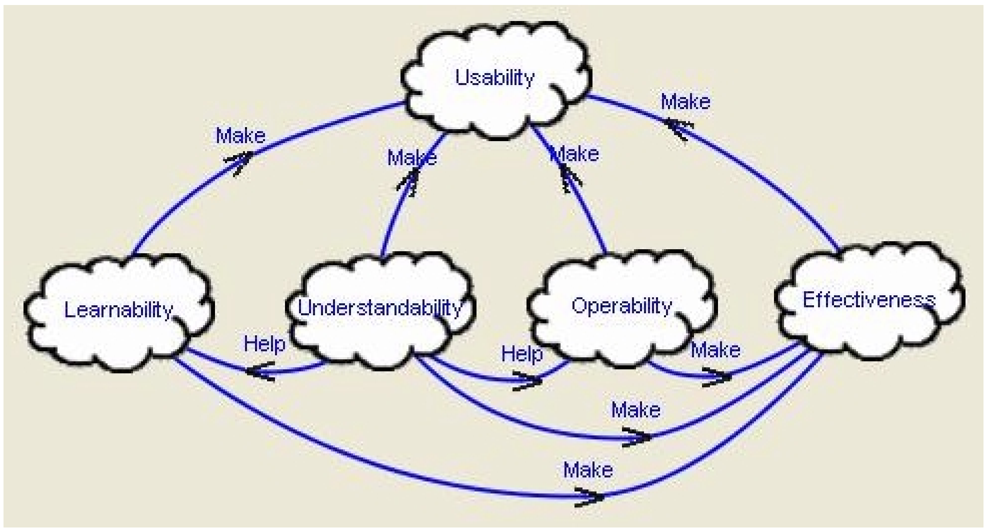


Figure 5-53: Usability decomposition

Some aspects of functionality, reliability and efficiency will affect usability. The relationship between those characteristics and usability can be positive or negative. For example, a system with such a bad time behaviour that it reacts too slowly for the user expectations will have a bad operability (thus a bad usability), but the use of assistants to improve the usability of the system will usually have a negative effect in the efficiency of the system, especially if the multimedia abilities of the system are used.

In order to understand the complexity of usability, there must be taking into account that it’s possible that there are different profiles of users and skills and attitudes among them, there are tasks of different complexities and the measures of success and interpretations of difficulty can be very subjective.

What is called usability in the Boehm’s model [Boe78] corresponds to what is called operability in this model. This characteristic has exactly the same name (usability) in the McCall’s model [McC94].

5.3.6.3.1 Learnability

The learnability of the system is the user’s effort for learning the use of a system, including the degree to which the software assists in lightening the learning process of new users. The learnability of the system doesn’t deal only with the effort for learning, but also for the effort for relearning features barely used. The five factors that have effect on the learnability of the system can be seen in Figure 5-54.

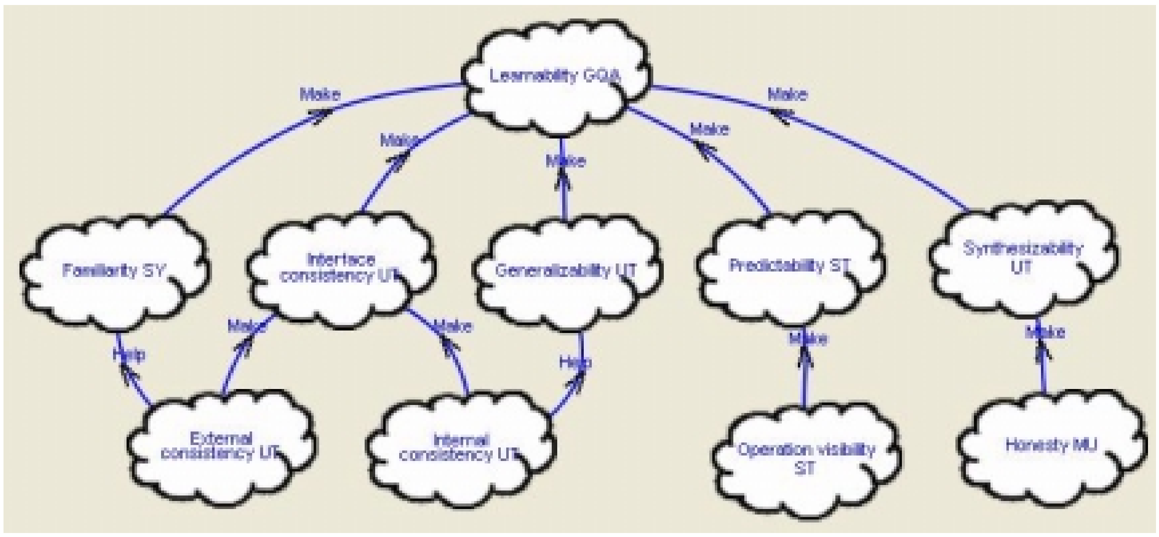


Figure 5-54: Learnability decomposition

If the user can predict the consequence of a certain action in the system, he will learn how to use it much faster. That means that the system should have some “Predictability”. When a user does an action, he’s not pursuing the action itself (for example, pushing a software button with the label “Print”), but the final operation as a consequence of the action (for example, printing the actual document). Therefore, something very important for the predictability of the interactions with the system is the “Operation visibility”. If the user has some information about the operations that can be performed, then he’ll be able to predict the consequences of his actions. It’s not only important that the user can identify the operation that will be executed as a consequence of a certain action on the system, but also all the other consequences of the action. This attribute could be called “Collateral effects visibility”. If the user can see what the consequences of an action are, he’ll learn when to make that action in order to take advantage of the desired collateral effects or to avoid the not desired collateral effects.

The “Synthesizability” of the system is another important factor of the learnability of the system. Synthesizability is the ability of the user to assess the effect of past actions on the current state of the system. The user should be able to know the current state of the system (reached as a consequence of the action) in order to be able to make his own mental model. The most decisive aspect for the synthesizability is the “Honesty”. If the system gives automatically information about the current state of the system, then it has immediate honesty. If this information is not given automatically, but the user can ask for it, then the system has eventual honesty. Let’s see an example: a drawing application in which the setting of points (to draw a line, for example) can be forced or unforced to some points of a grid. If a user, who doesn’t know the goal of a button to force the points to be in the grid, presses it and no message is shown (eventual honesty if the user can check the state of the grid or no honesty if he can’t check it), the user will discover the new state when he tries to set a point. If he has done some other things in between, it will be quite difficult for him to assess the action that produced that effect. One case in which immediate honesty is essential is in the communication of errors with the user inputs. The user should know what he’s done wrong in order to be able to avoid erroneous behaviours in the future.

The “Familiarity” of the user with as much concepts used in the system as possible is very important for the learnability of the system. If the system uses concepts based on the knowledge that the user is supposed to have, the system usage can be faster learned by the user. Another way to take advantage of the familiarity of the user with certain concepts is the use of standards. For example, when a user who has some familiarity with applications for Microsoft Windows wants to copy something written in a document to another part of the document, he will try to use the typical Copy&Paste operations, so common in the applications for Microsoft Windows. The additional use of the shortcuts Ctrl+C for Copy and Ctrl+V for Paste (what could be considered a de facto standard) will dramatically improve the learning of the use of the system. The user doesn't have to learn these features of the application, because he's used to them and he can concentrate on learning the unknown features. Don't surprise the user, don't use things in other way than the most usual one (the one the user will be probably most used to). For example, don't use a gear as an icon for a button to run an application, because that icon can be usually seen on buttons to configure things (Settings).

The "Generalizability" of similar ideas into one concept is also quite important, because once learned the concept, all the associated ideas are also learned. It means that if there is a general way of doing similar things, the user learns much faster because of the association of ideas and applies previous interactions to similar situations. Let's see an example within a suite of applications: when a user who knows how to use a certain application of the suite tries to use another one, he will look for all the common buttons (such as open, save...) in the same or similar situation as in the already known application.

If these last two factors are analyzed, it can be seen that they bear on taking advantage of the experience of the user and of his analysis capability. The best has to be taken from the experience of the user with other systems (familiarity) and with the system itself (generalizability). The "Interface consistency" of the system is what makes it possible. The consistency can be decomposed in two different aspects: internal and external consistency. "Internal consistency" means that similar situations that are given inside of the system show similar behaviours. For example: all the error messages of an application are communicated with a certain sound. The internal consistency of the system helps to have a system with a good generalizability. "External consistency" means that a situation that is given inside of the system shows a behaviour similar to that of similar situations given in other systems. For example: the shortcut for Copy in an application for Microsoft Windows should be Ctrl+C in order to be consistent with that used in most applications for Microsoft Windows. The external consistency of the system helps to have a system with a good familiarity.

In the McCall's model [McC94] there is a factor called "Training" which is similar to this subcharacteristic. Nevertheless when the description is carefully read, it can be seen that it isn't exactly the same: "Those attributes of the software that provide transition from current operation or initial familiarization".

5.3.6.3.2 Operability

Every software system is a tool to do something and the user has to be able to do that thing as easy and well as possible. In other words, the effort of the user for using the system and using it properly has to be as small as possible. This subcharacteristic is decomposed as it can be seen in Figure 5-55.

When a user operates a user-interface, the operating time comes partially determined by the design of that interface. For example, there is a minimum time that a user need to fill three fields up to fulfil a task, this minimum time would be less if the same user would have just to click a button. This time determined by the design of the interface will be called "Usage time" and has been already mentioned in the description of time behaviour (see efficiency), because of its big influence on the overall efficiency. This usage time comes mainly determined by the operability of the system.

Let's suppose that a user has to control a system in which the user only has to react to some events of variable and/or undetermined period (which could be as long as half an hour). Will the user control the system properly? He'll probably loose concentration from time to time. The "Communicability" of this system is very important for its operability. In general, every system has to communicate the user all the things that can be useful for him to control the system. And the clearer the message for the user is, the better he'll control the system. One important aspect of the communicability of the system is the "Feedback" that it gives to the user, so that he can control the system and operate it in a proper way. Another important aspect is the "Failure explicitness" of the system. The system has to communicate the user all the states detected in the system to which he/she has to react: failures in the use, errors due to an internal error of the system, dangerous states in a controlled system…

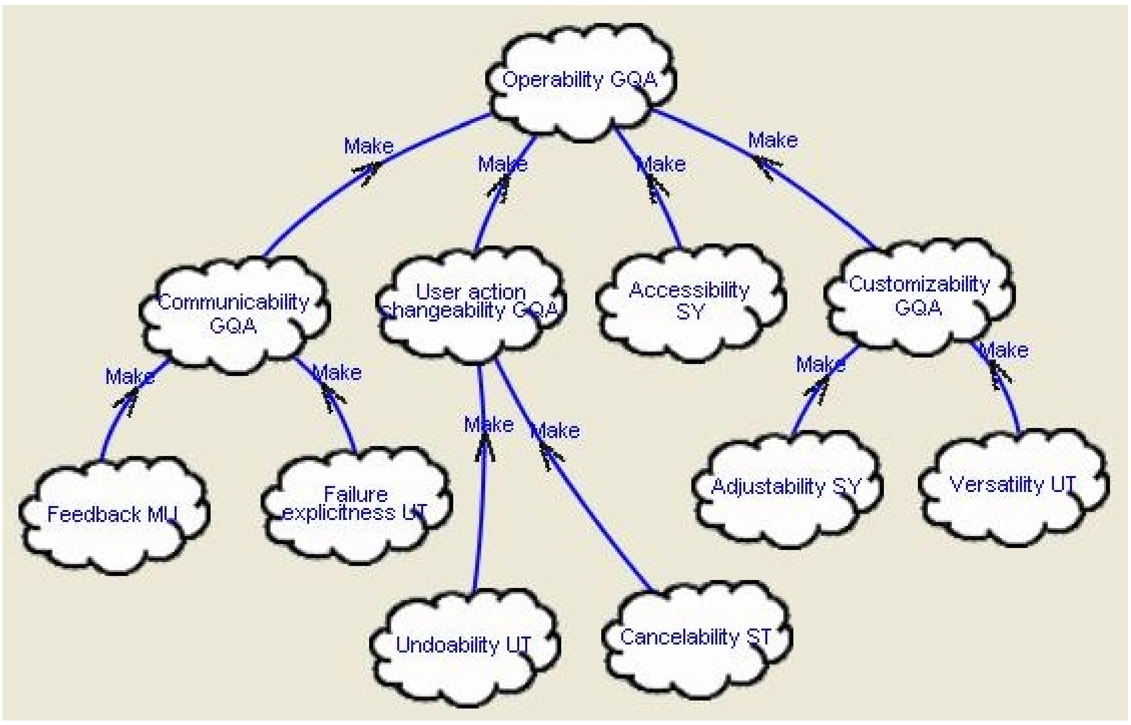


Figure 5-55: Operability decomposition

There is no perfect user; therefore some users do errors in the introduction of data, selection of options, etc. Can it be acceptable that a user during a 20 steps process can't undo the actions? If he did an error during the 20th step of the process, should be user repeat all of them again? The "Undoability" of the system in this process is very important. What happens if the user makes an error during the 1st step of the process and he can't cancel it? The "Cancelability" of the system in this process is also very important. The "User action changeability" of the system (decomposed into undoability and cancelability) allows the user changing his or her actions (sometimes errors).

The user of a software system should be able to adapt it to his/her own pleasures as much as possible. Or the system should provide at least some possibilities to do things (when it's possible). The "Customizability" of the system allows the user to adapt the system to him/her, not that the user has to adapt to the system. The "Adjustability" of the system is the possibility for the user to adapt or adjust the system to his/her pleasures. The "Versatility" of the system is what allows the user to select the most comfortable way (if there is more than one) for him/her to do something. For example, to select a path the user should be able to write it directly or to look for it in a tree structure. The customizability of the system is usually good for the operability of the system, but sometimes it (especially the adjustability) can be negative if the user does a misuse of it. For example, a user of a system without experience would get the operability of the system worse if he configured it for an advanced user.

The "Accessibility" is the effort of a user to operate the system when he or she is not a typical user (non-typical users are handicapped persons, kinder, elders…) or the context situation is not the typical one (wearing protection glasses, gloves…).

5.3.6.3.3 Understandability

This subcharacteristic is very important in no-normal situations in which confusion may be high and mistakes may be regrettable. It is quite related to the two previous ones: learnability and operability. The easier the user recognizes concepts in the system, the faster the user learns them. Once learned, if they can be easily understood, they will be easily used. In other words: a system that can be easily understood can be easily learned and operated. This subcharacteristic can be decomposed as it can be seen in the following figure (Figure 5-56).

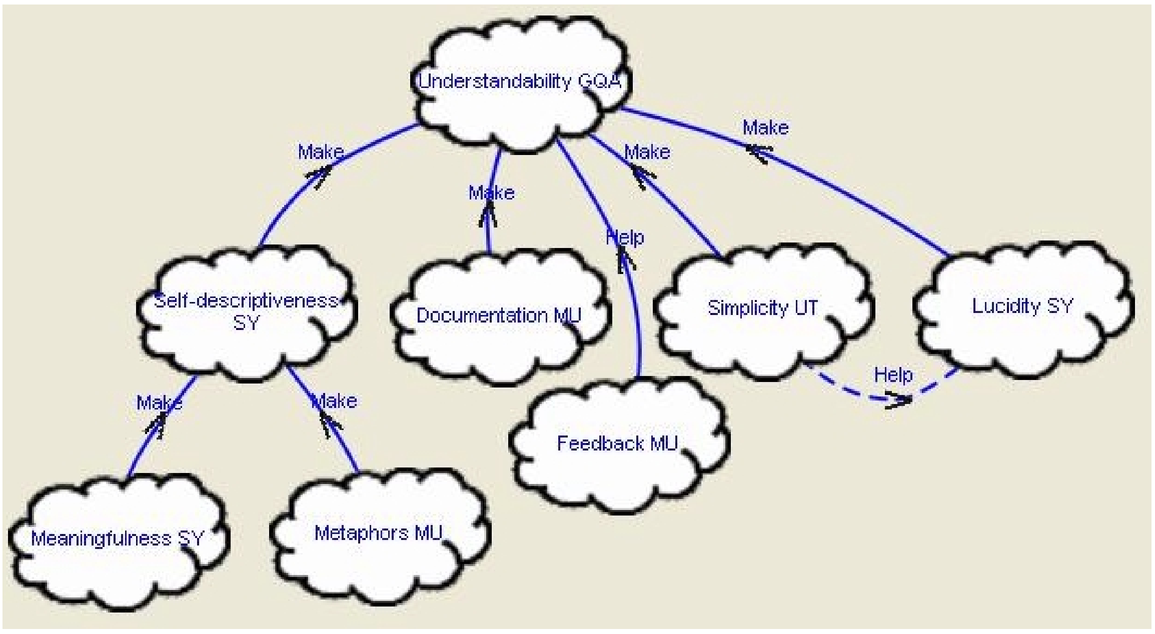


Figure 5-56: Understandability decomposition

In order to have an intuitive system the most important thing is its "Self-descriptiveness". The "Meaningfulness" of the names and abbreviations used is very important. The text "E" in a button to exit the application is meaningless. A meaningful name would be "Exit". "Metaphors" (especially real-world metaphors) can be helpful, but they should be used considering very carefully the knowledge that the user can have of the things on which the metaphor is based. The use of a floppy disk as icon of a button to save a document will be a good metaphor if the expected users have familiarity with computers with graphical interface, but bad if the expected users haven't familiarity with those things. A self-descriptive system will be more predictable, thus improving the learnability of the system.

The “Documentation” of the system with on-screen instructions, diagrams, help documentation, user guide, assistants… is very important in order to replace the lack of self-descriptiveness. Not everything can be self-descriptive, that’s why some documentation is almost always necessary, although not explicitly specified as a requirement. A good documentation of the system improves also the predictability of the system (see Learnability).

The optimal situation for the understandability is when the user understands a priori the concepts involved in his/her interaction with the system; but sometimes it doesn’t happen. In these cases it’s much better if the user can understand them after the interaction. The “Feedback” of the system helps the user to understand them after the interaction.

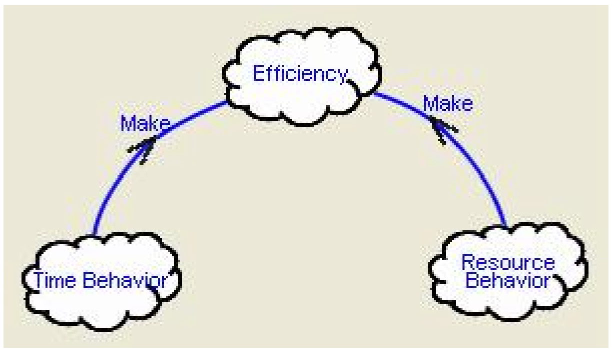
The simpler the system is, the easier to understand it will be. The “Simplicity” of the system is one aspect that, well used, is very important for the understandability of the system; but that, bad used, can get the understandability of some features or users worse. Let’s see an example: the advance features of a system will usually worsen the usability of the system for basic users. It’s a habit hiding this advance features in order to improve the usability of the system for basic users; but if they are too difficult to be reached by the advance users, the usability of the system for them will be seriously deteriorated. The KISS principle (“Keep It Simple, Stupid”) well applied can drive to very understandable systems.

When a person tries to understand something new, that person first try to get an overview of that thing and then refines its comprehension. This behaviour pattern has to be taken into account to improve the understandability of a system. The “Lucidity” of a system is how easy is it for the user getting an overview of the user-interface. The already mentioned “Simplicity” will usually improve the lucidity.

Sometimes the word comprehensibility is used as a synonym of understandability. What is called understandability in the Boehm’s model [Boe78] is the addition of the subcharacteristics learnability and understandability. The equivalent factor in the McCall’s model [McC94] is called communicativeness and this is its definition: “Those attributes of the software that provide useful inputs and outputs which can be assimilated”.

5.3.6.4 Efficiency

Figure 5-57[3] shows the basic decomposition of system efficiency. The two softgoals (“Resource Behaviour” and “Time Behaviour”) are results of the ISO 9126 specification and they are covered in more detail on chapters 5.3.6.4.1 and 5.3.6.4.2.



[3] In the decomposition examples of this chapter the relationships between softgoals will almost always be “Make”. Further relationship types will have to be used when applying the decomposing patterns on real-world applications.

Figure 5-57: Efficiency decomposition

There are three principles that help the developer analyze efficiency and to build efficient systems:

Centring Principle: Efforts to improve performance should focus first on the *critical* and *dominant* parts of the workload and the system.

Processing vs. Frequency Tradeoff Principle: Minimize the product of the processing time per execution of an operation, and the frequency of executing the operation.

Fixing Principle: For good response time and reduced resource utilization, “fixing” (the process of mapping functions and information to instructions and data) has to be done as early as possible. Note that there is a tradeoff between these two softgoals (early fixing) and increasing flexibility (late fixing).

5.3.6.4.1 Time Behaviour

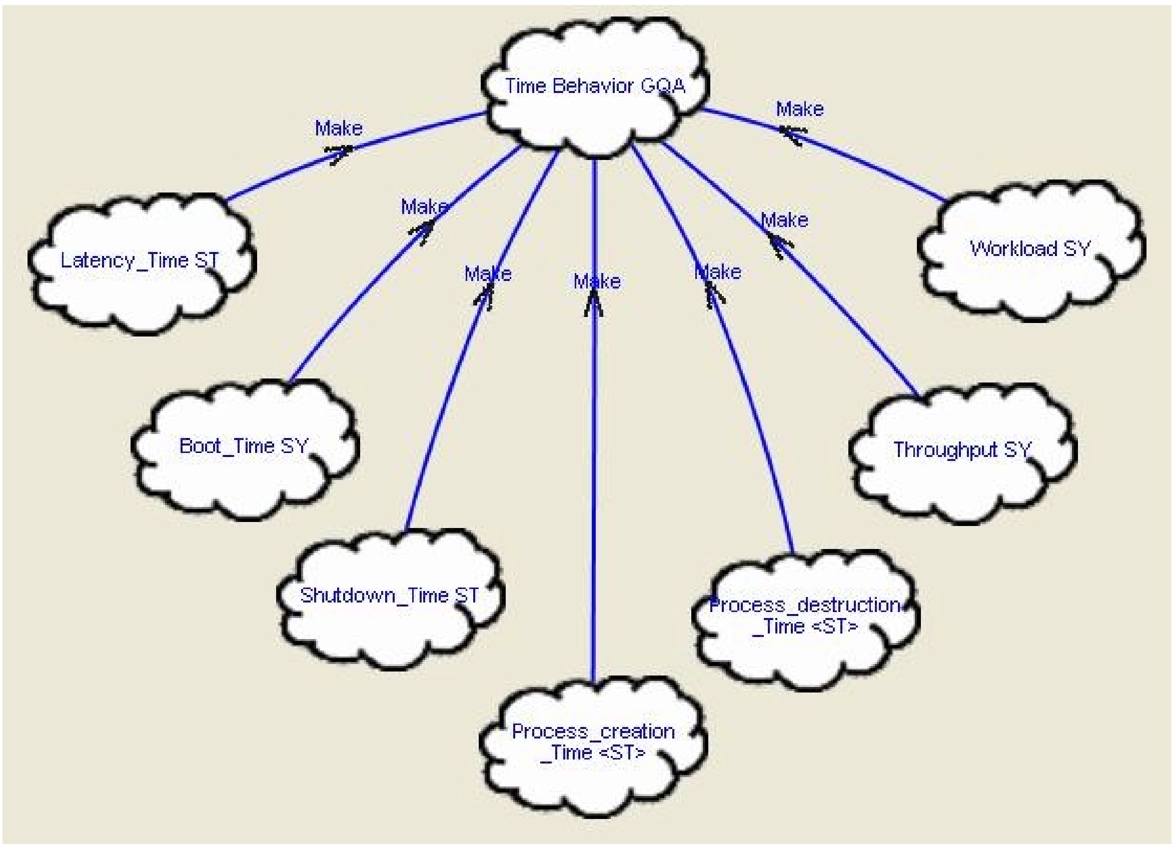


Figure 5-58: Time Behaviour decomposition

Figure 5-58 shows a basic decomposition pattern for “Time Behaviour”. The softgoals “Workload” and “Throughput” refer to the “processing capacity” of the system, the first one refers to the processing itself and the second one to the transmission of data (network, flash, etc.). They can be easily analyzed using a physical decomposition pattern along with the “Centring Principle”.

Although it is not shown in Figure 5-58, two different system statuses have to be covered regardless the type of time aspect we focus on: Peak Status and Off-peak Status. That means, for instance, that there will be two extra softgoals called “Process Creation Time-Offpeak Status” and “Process Creation Time-Peak Status”, covering the elapsed time to create a new thread or task when the system load is average (Off-Peak) and when it is hard (Peak). Once again, this refinement has to be done for every single type of time aspect we analyze[4].

There is another aspect (not shown in Figure 5-58) that affects the time behaviour of the system, the time that the user has to invest in interacting with the system, what will be called “Usage time”. It depends on the operability of the system (therefore it will be there modelled), but it will have a big influence in the efficiency of the system. For example, an interface in which the user can select a colour from a list of the three only possible colours will be much more time-efficient than another one in which the user has to write the complete name of the colour.

5.3.6.4.2 Resource Behaviour

Figure 5-59 shows a basic decomposition pattern for the NFR “Resource Behaviour”. We first decompose it into the system's own resources behaviour ("Resources [System]") and additional peripheral's resource behaviour ("Resources [Peripherals]"). This last softgoal addresses third-party entities that our system only uses part-time, and their decomposition will entirely depend on the hardware of our system. For instance, in the example of Figure 5-59 we have split “Peripherals” into three generic softgoals[5]:

"Resources [Floating Point Units]" (it could also be a subpart of the system itself and not considered as peripheral)

"Resources [Cache Units]"

"Resources [I/O Channels]"

On the other side, the softgoal "Resources [System]" is divided into "Physical Resources [System]" and "Logical Resources [System]". The first one refers to the system's most important physical resources, such as main memory (which could be also refined into ROM, RAM and so on), secondary storage and processor utilization. Three extra softgoals have been included to cover those topics. The softgoal "Logical Resources" covers the software logical decomposition as "Logical Resources [System Data]" and "Logical Resources [System Code]" reflect.

Finally “Workload distribution” is only valid for systems which distribute the execution (distributed systems or multiprocessor systems) or the storage of data (in different flash modules to accelerate the writing process, for example). It addresses the appropriateness of that distribution.

[4] It is not shown in Figure 5-58 because of readability purposes.

[5] Other decomposition approaches may also be applicable depending on the hardware configuration or on the hardware usage profiles.

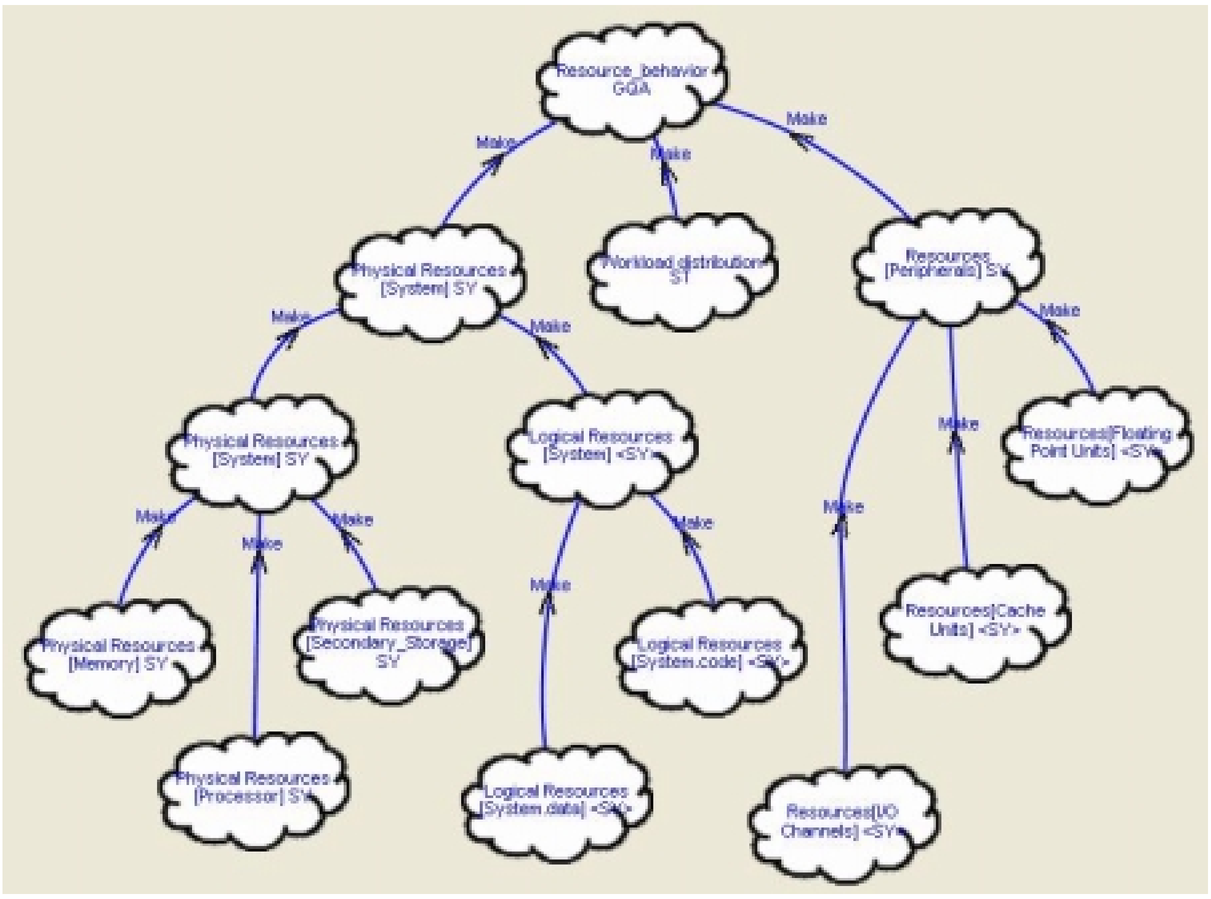

Resource_behavior GQA
Make
Make
Make
Physical Resources [System] SY
Workload distribution ST
Resources [Peripherals] SY
Make
Make
Make
Physical Resources [System] SY
Logical Resources [System] <SY>
Make
Resources[Floating Point Units] <SY>
Make
Make
Make
Make
Make
Make
Physical Resources [Memory] SY
Physical Resources [Secondary_Storage] SY
Logical Resources [System.code] <SY>
Resources[Cache Units] <SY>
Physical Resources [Processor] SY
Logical Resources [System.data] <SY>
Resources[I/O Channels] <SY>

Figure 5-59: Resource Behaviour decomposition

5.3.6.5 Maintainability

Figure 5-60 shows the basic decomposition of system maintainability. The first four softgoals (“Changeability”, “Testability”, “Analyzability” and “Stability”) are results of the ISO 9126 specification and they are covered in more detail on chapters 5.3.6.5.1, 5.3.6.5.2, 5.3.6.5.3 and 5.3.6.5.4.

The fifth one (“Reusability”) covers the attributes of software that bear on its potential for complete or partial reuse in another product. It will be covered on chapter 5.3.6.5.5.

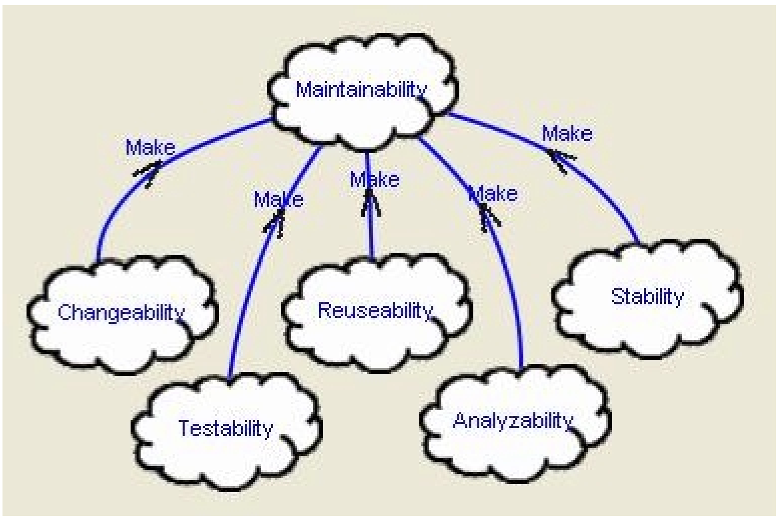


Figure 5-60: Maintainability decomposition

It is important to distinguish between maintenance and maintainability. Software maintenance can be defined as “the continuous process of keeping the program running, or improving its characteristics”. Maintainability is “the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements”. Although different, both terms are obviously related, and on both terms four different aspects can be identified: Corrective aspects (those who deal with identification and correction of software faults), Adaptive aspects (to adapt software to functionality or environment changes), Perfective aspects (those who are related to any sort of enhancements), and Preventive aspects (changes for future and possible expansions). We will cover these aspects in the following points.

5.3.6.5.1 Changeability

Figure 5-61 depicts a basic decomposition pattern for “Changeability”. The best way to assess changeability is to analyze the impact of changes. This tasks evolves two actions: First, identify the components of the system that will be “touched” by the changes; and secondly, find out to what extent those systems will be affected.

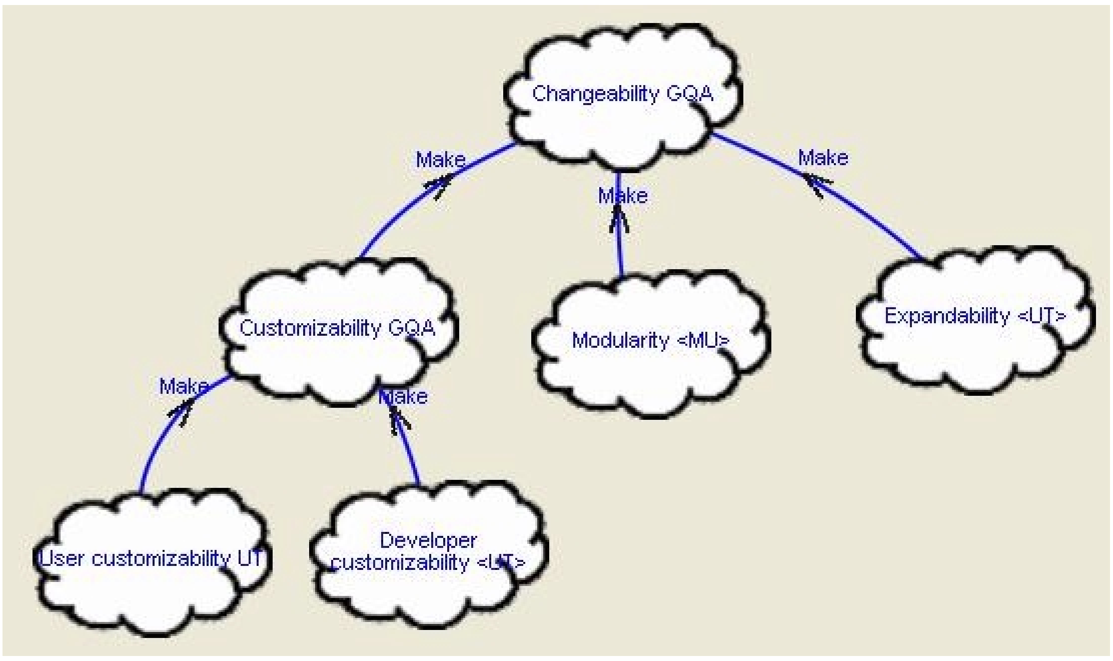


Figure 5-61: Changeability decomposition

Customizability covers all kind of parameter-driven changes. It is necessary to distinguish two aspects under Customizability: User Customizability (parameters that are given to the user in order to adapt the system to their necessities) and Developer Customizability (parameters that are hidden for the user, but available for the developer to adapt the system to new customers or new functioning frameworks.

Modularity is closely related to one of the cornerstones of modern software design: information hiding. Software modularity is the practice of functionally dividing a software system into individual modules, which can then be designed and implemented independently, and later integrated into a cohesive solution. All interactions between modules are made through well-defined APIs, ensuring that modules can continue to be developed independently.

Expandability can also be called “scalability”, and it mostly deals with the preventive and perfective aspects of changeability. Questions such as “will the software system considerably grow in size or number of external dependencies?” or “Is the software ready to provide network capabilities in the future?” characterize the typical aspect of Expandability. Scope topics like going from a single-user to a multi-user environment are also covered by this softgoal.

Uninstalling the system before the changes and installing the system with the changes can be necessary because of the nature of the changes committed, but those aspects are evaluated in a portability subcharacteristic: “Installability”.

5.3.6.5.2 Testability

Figure 5-62 depicts a basic decomposition pattern for “Testability” in which further refinement can be easily done applying a HW/SW sub-system decomposition approach. Testability is the degree to which a software system facilitates testing in a given test context, and therefore it is directly related to test effort reduction.

The softgoal called “Self Test Capabilities” covers all topics regarding dynamic, built-in, self test facilities that the system offers. These tests are normally carried out without (or with minimum) human interaction, are (almost) transparent to the user and run in parallel to the normal system function. They handle error identification on run-time, and can trigger further “Fault Tolerance” activities on the system. Automatic facilities to identify possible causes for poor network or server performance, or watchdog-logging capabilities that the system performs in order to make internal states easier to follow for the user/developer are typical examples.

The “Test Suite” softgoal addresses aspects like testing interfaces that the system offers to third-party testing tools or remote debugging capabilities. The main difference with Self-Test Capabilities is that no automatic or built-in facilities are covered here: a special test interface that can be used for someone else’s program is the typical utilization example for this topic. Depending on the scope (the user’s testing interface does not necessarily have to be the same as the developer’s one) it is possible to distinguish between “User Test Suite” and “Developer Test Suite”.

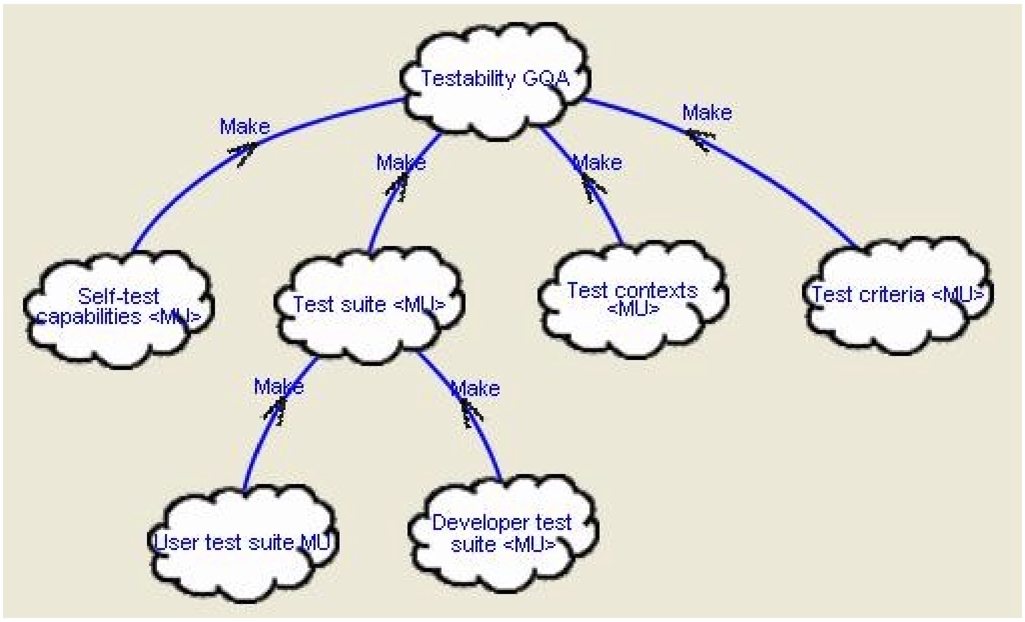


Figure 5-62: Testability decomposition

There is no absolute measure of quality: software quality is always defined in terms of particular customer needs. Something similar can be said for testability: Whether a component or system is testable or not depends on the context and that is exactly what the softgoal “Test Context” addresses. The context relevant for testability assessment includes:

Test constraints

Intended use of the component

Test tools

Finally, depending on the test criteria applied we have to address different issues of design for testability. A reason for this is that test efficiency is less sensible to characteristics of the program code when specification based test criteria are used then if program based test criteria are used. This field is covered by the “Test Criteria” softgoal.

5.3.6.5.3 Analyzability

Figure 5-63 depicts a basic decomposition pattern for “Analyzability”. The three softgoals cover all the different aspects to take into account when improving the analyzability of a system.

The first softgoal addresses the “Simplicity” of the system. A simple system can be easily analyzed than a complex one. The simplicity can not be easily evaluated, but it is quite easy evaluating it as the opposite of complexity, and many metrics to measure the complexity of a software system can be easily found.

The second softgoal is the most important one; this is the “Understandability”. A system can be analyzed only if it is well understood by the person who tries to analyze it. There are five aspects that can be analyzed in order to evaluate the understandability of the system, they are next analyzed.

The first and second aspects (“Name Conventions” and “Documentation”) cover not only what their names refer to, but also topics as keywords, name libraries, document format, and further notation rules. The use of this rules and guidelines make the whole process homogeneous from the notation and documentation point of view, avoiding misunderstandings and communication-media breakdowns.

The third one (“Description and Programming Languages”) addresses the different aspects on how the languages used along the development process influence the understandability of the system. For example, the use of languages like UML or SDL during the analysis and design phases makes the system much more understandable and easy to follow. Regarding the coding phase, object-oriented and structured languages offer a wide variety of artefacts to improve understandability in comparison to assemblers and other similar languages.

The forth one (“Analysis/Design Paradigm”) is deeply related to the previous one, as the type of paradigm (“spaghetti code”, structured, object-oriented, extreme programming, etc) not only will have a huge influence on the understandability (the basics of object-oriented paradigm such as encapsulation, inheritance and polymorphism are well know for their ability to make systems more understandable), but also will determine the languages to be used in the development process. Some OOA-OOD principles that can be used to improve system analyzability are the following:

Open-Closed Principle (OCP): A module should be open for its extension and closed for its modification.

Substitution Principle (SP): The subclasses must be substitutable by their base classes.

Dependency Inversion Principle (DIP): Depend upon abstraction. Do not depend upon specifications.

Interface Segregation Principle (ISP): Many clients’ specific interfaces are better than one general purpose interface.

Default Abstraction Principle (DAP): Introduces an abstract class that makes the implementation in default of most of the interface operations between the interface and the class that implements it.

Interface Design Principle (IDP): “Program” an interface, not an implementation.

Black Box Principle (BBP): Favour the object composition over class inheritance.

Do not Concrete Superclass Principle (DCSP): Avoid maintaining concrete superclass.

The fifth and last aspect ("Use of Patterns") is also related to the two first. Software patterns are general, well known design and architectural solutions that can be applied to solve particular problems under particular circumstances. Their use makes the system more intuitive and easy to understand as software patterns are well known all over the programmers' community and are better recognized than tailor-made, user-specific solutions.

The third softgoal is the "Structuredness" of the system. The use of patterns in the design of a system will always make it more structured; anyway patterns have to be used only when they suit. A bad used pattern will make a system more structured, but it will usually increase its complexity and can have a bad influence in other characteristics of the system (for example in the efficiency).

The forth softgoal ("Backwards traceability") addresses the effort needed to trace the history of the execution of the system. It is important not confusing this aspect with what is usually called "Traceability" or "Forward traceability", what addresses the effort needed to know which parts of a system are consequences of a requirement. The backwards traceability of a system eases understanding who behaves a system during its run-time.

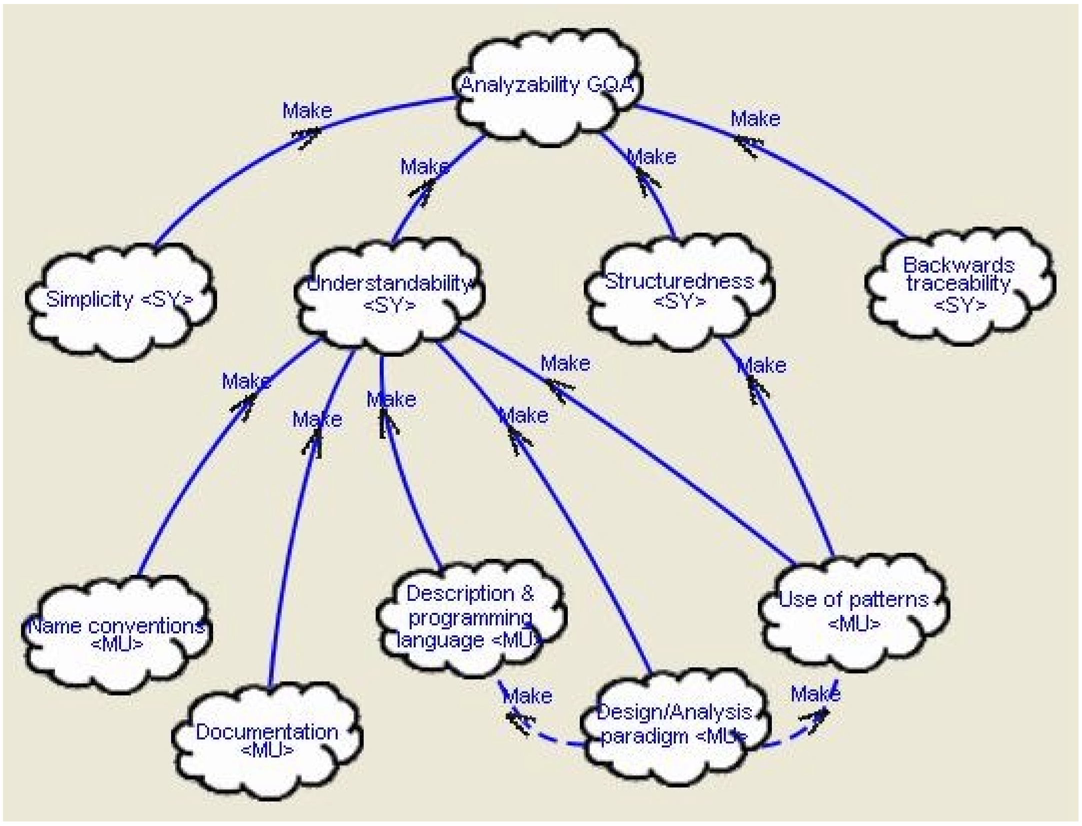


Figure 5-63: Analyzability decomposition

5.3.6.5.4 Stability

Figure 5-64 depicts a basic decomposition pattern for "Stability". Stability can be defined as a condition where features are implemented, integrated, and tested to a planned degree of completion, combined with low rates of recent change. The combination of planned *degree of completion* and lack *of change* is crucial; things that are completed -- including testing -- but have recently changed considerably should not be considered stable. Conversely, incomplete things that have not changed in a long time are not really stable, due to incipient change. Also note that something does not have to be in its end-form to be stable -- otherwise, the only stable point to exploit would be at project end! Alternating incremental cycles with iterative cycles can provide stable points even during evolution.

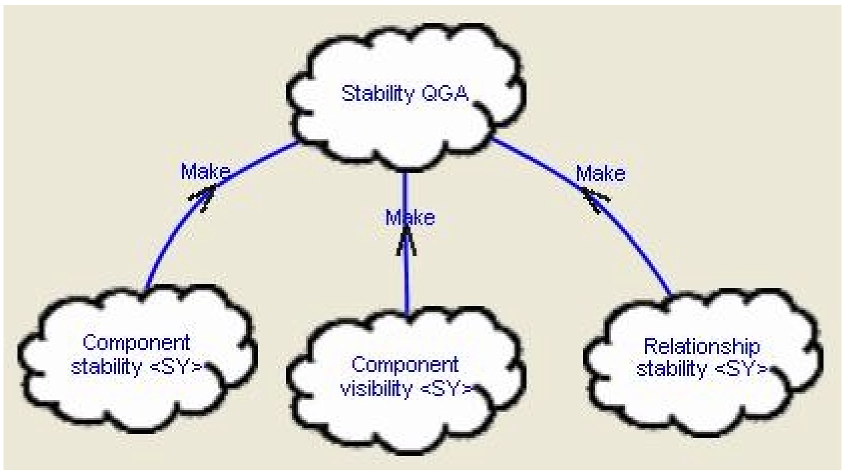


Figure 5-64: Stability decomposition

As the picture shows, the analysis pattern of “Stability” is rather simple and straightforward, based on a sub-system decomposition approach. The stability of a software product is mostly affected by the stability of its components (softgoal “Components stability”) and the stability of the relationships among the components (softgoal “Relationship stability”). These two softgoals can be easily refined through further sub-system-based decomposition analysis.

Additionally, assuming that the fundamental issue in communication is that some sort of visibility must exist among the components, visibility implies dependency with respect to changes, and hence it has consequences on the entire system's stability. The third softgoal (“Component visibility”) addresses this aspect.

5.3.6.5.5 Reusability

Reusability is defined as those attributes of software that bear on the potential for complete or partial reuse in another software product. There are two conditions so that an element of a software system can be reused: that it can be analyzed and understood and that it’s suitable for the new use.

Depending on who the customer of the system is and what’s been contracted, this attribute can be very important or useless. If a software developing company order some libraries (with source codes) to another software company, then this attribute can be very important for the customer, because he’ll be able to take advantage of the reusability of the system. If the customer is a company which wants only the final product and will never deal with software developing, then this attribute is almost useless. This characteristic is not only important for the system itself (some parts of the system can be reused when developing some other parts of the same system), but also for future systems (what mainly benefit the development team).

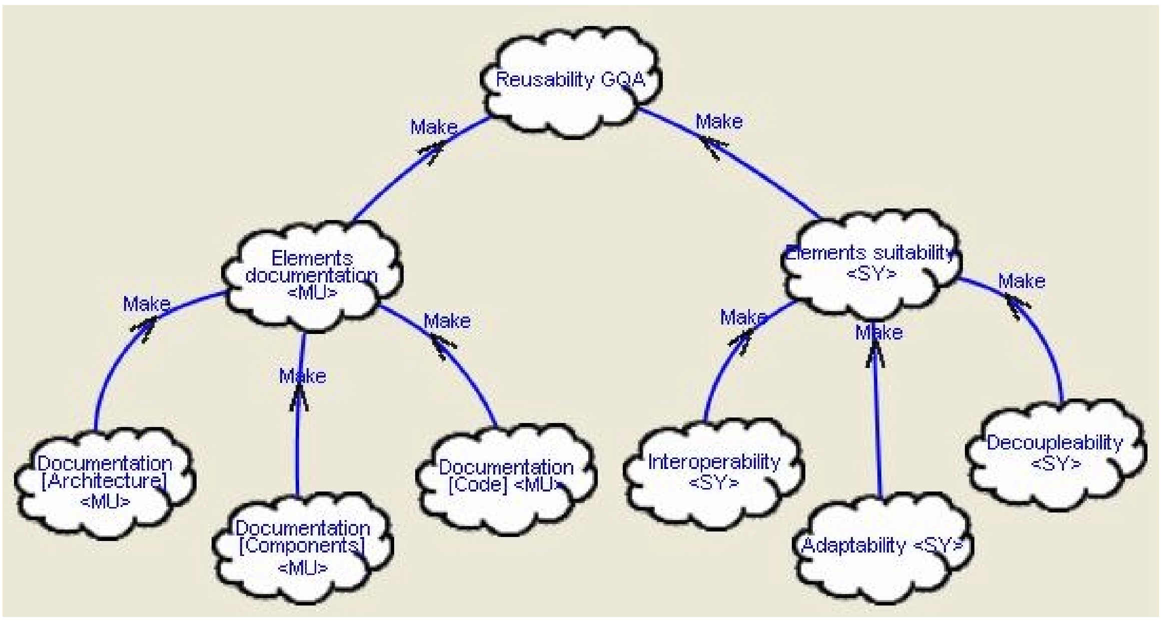

Reusability GQA
Make
Make
Elements documentation <MU>
Elements suitability <SY>
Make
Make
Make
Make
Make
Make
Documentation [Architecture] <MU>
Documentation [Components] <MU>
Documentation [Code] <MU>
Interoperability <SY>
Adaptability <SY>
Decoupleability <SY>

Figure 5-65: Reusability decomposition

Figure 5-65 depicts a basic decomposition pattern for “Reusability”.

Reusability can be an attribute of all the elements of a software system (architecture, components, code…) and it’s desirable for all its elements.

Software developers very often develop things already existing (sometimes even developed by a working college) and that are available for him/her; but there are many difficulties for the reusability of software: difficulties to identify reusable software, difficulties to get the information necessary to use the software reused (especially because of a lack of documentation), too much effort needed to reuse the software and some others. A system shouldn’t have these difficulties to be reusable, because if some software already used and tested can be reused, the resulting software will be more reliable and the development costs will be reduced. Some documentation of the software (what could be called “Elements documentation”) is necessary so that the developer can evaluate the reusability of its elements and, if it’s reusable, so that it can be used.

In order to have a system with reusable architecture, the “Architectural documentation” is the most important thing. One of the best ways to write that technical documentation is the use of UML tools.

For the components reusability, the “Technical documentation” is the most important things. As for the architectural documentation, for the components documentation the use of UML is very useful.

If the code of the system was written following some “Style conventions” and with a good “Code documentation”, then it will be more reusable. For example, in order to be able to reuse a function it’s very important that its interface (parameters, returned values…) is well documented.

The parts of the software that can be analyzed and understood are potentially reusable parts. Those that are suitable for the certain use can be reused. This attribute could be called “Elements suitability”.

There are three attributes of the software that will improve the elements suitability. When reusing some software elements, the elements reused have to be somehow adapted to the new system. That's why the "Adaptability" (subcharacteristic of Portability that deals with the effort necessary to adapt a system to a new environment, see chapter 5.3.6.6.1) of the system will help to make some elements suitable for the new use. The reused elements are usually somehow encapsulated and they have to interface with elements of the new software. Therefore the "Interoperability" (subcharacteristic of Functionality that deals with the interface abilities of the software, see chapter 5.3.6.1.3) of the system is the other attribute. These two attributes will be studied in posterior chapters.

The "Decoupleability" of the components it the third attribute for the suitability of the components for the reusing. The weaker the coupling between components is, the better their reusability is.

This subcharacteristic is one of the factors (with the same name) which compose the McCall's model [McC94].

5.3.6.6 Portability

Within the expression "Operating environment" many different things have to be considered. As already said in the definition, hardware and software are two important things in the environment of the system. Still the organizational environment should be included in this operating environment.

An appropriate use of modularity (in order to isolate the environment-dependent components) will improve the portability of the system in general. All the portability subcharacteristics will profit from the environment independence that can be reached by means of the already mentioned isolation of the environment-dependent components. However, as it will be seen below, the modularity of the system is especially important for the adaptability of the system.

The improvement of the portability of a system usually implies an increment of the complexity of the system. This increment of the complexity has a negative effect on the efficiency, usability and maintainability of the system; but can have a positive effect on its flexibility, interoperability, reusability and testability.

This subcharacteristic is divided in four subcharacteristics: adaptability, installability, replaceability and co-existence. This decomposition can be seen in Figure 5-66

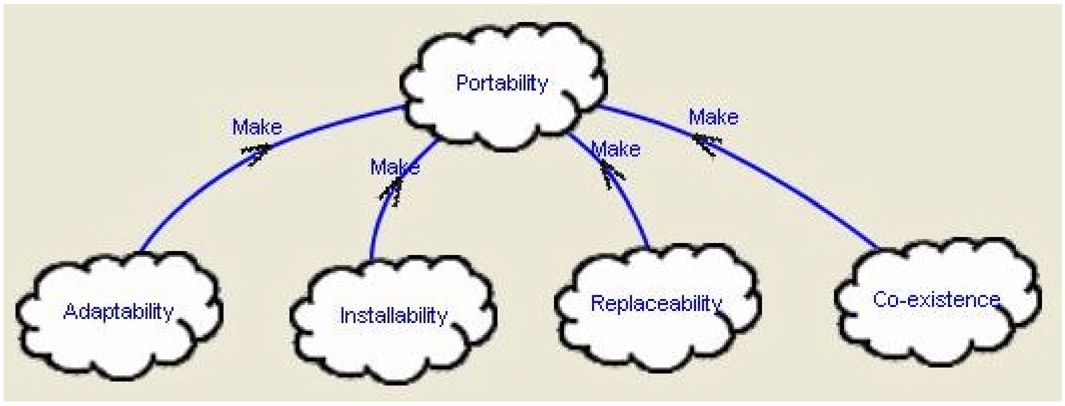


Figure 5-66: Portability decomposition

This characteristic is also considered an important quality attribute in other quality models. In the McCall's [McC94] model the portability of the system is one of the eleven factors that describe the quality of a software system. In the Boehm's [Boe78] model the portability of the system is one of the eight software characteristics.

5.3.6.6.1 Adaptability

This subcharacteristic can be divided in the six factors that can be seen in the following figure (Figure 5-67).

The internal capacity of the system should be scalable depending on the resources available for the system; this attribute is the "Scalability" of the system. For example, the size of the screens shown should be automatically scaled depending on the size of the display; the size of some files automatically saved by a system (whose size is big in comparison with the total memory) should be automatically limited depending on the memory available for the system.

Something already mentioned when talking about the operability of the system, its "Customizability", will be also positive for the adaptability of the system. The user is part of the environment of the system and the system should be able to adapt to changes of the user (a different user or the same user with different capabilities –after a course, for example-).

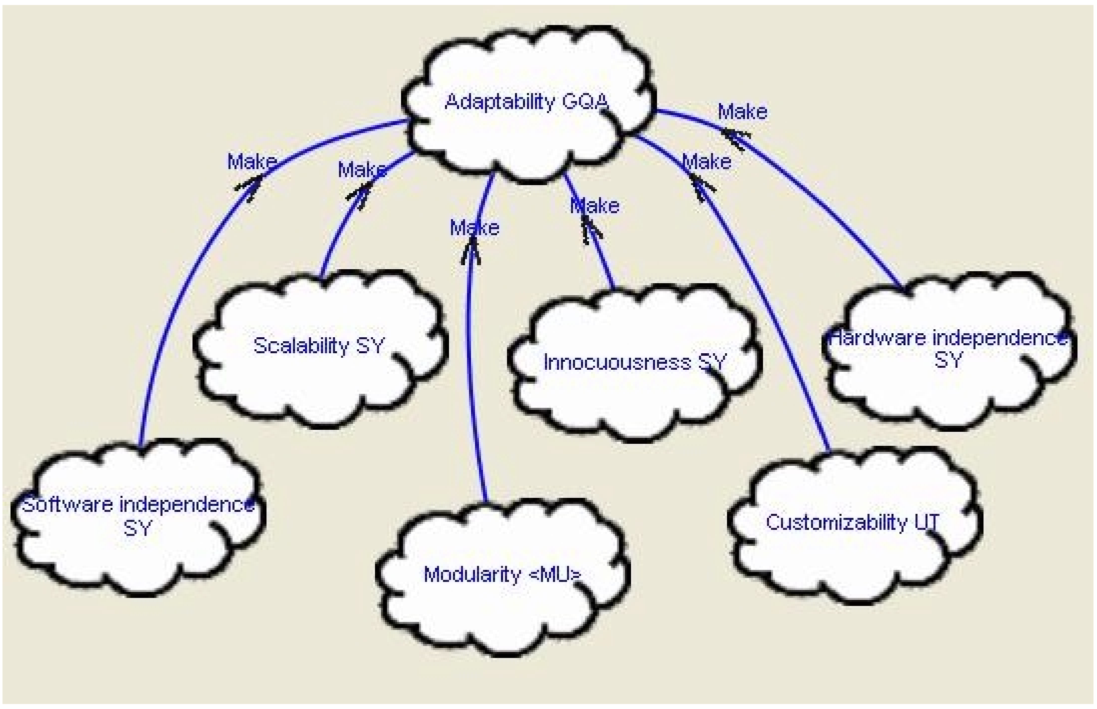


Figure 5-67: Adaptability decomposition

The "Modularity" of the components reduces the cohesion between components, what makes every change that could be necessary for adapting much easier. This factor is positive for almost all the other five factors of the adaptability of the system.

The system should modify the host system (the system in which it will be integrated) as less as possible (nothing if possible) and if it does it, shouldn't modify the original functionality, only extend it. The "Innocuousness" of the system for the host system is very important, because if the system damages the functionality of the host, some modifications will have to be done in the system in order to adapt it to the host. A very good example of non-innocuous systems: some applications for Microsoft Windows overwrite some DLLs already installed, but the functionality of the new DLLs is slightly different from that of the original ones and the applications that were using the original DLLs doesn't run properly anymore.

As much "Software independence" as possible is very important so that the system can be installed in one or other system without modification because of software dependencies. One factor that improves the software independence of the system is the use of "Standard protocols and interface routines" and "Standard data representations and structures".

The functionality of a system which could be running in a host with flash memory or a hard disk as persistent storage system shouldn't depend on the used storage system. Therefore, the "Hardware independence" of the system is also very important. A modular system with drivers and a layers structure will be more hardware independent than a monolithic system.

5.3.6.6.2 Installability

It can't be forgotten that there will always be an installer of the system. Depending on the experience of the installer, the installability can affect more or less the resulting suitability and operability of the system. For example, if the features of the system depend on the modules installed, the installer should be able to install those modules easily. In Figure 5-68 a decomposition of this subcharacteristic can be seen.

The "Eloquence" of the system during the installation is very important. Eloquence means that the information (written or online) is good enough for the installer to know the requirements of the system and the consequences of the installation options and that the system communicates the status of the installation process and shows a final summary so that the installer knows that if there were problems or everything was OK. The eloquence of the system is especially important for the maintainability of the system, because it permits eliminating installation errors as source of malfunction. For example, if a DLL has to be substituted by a newer version in order to able all the features of the system, but it can't because a running application uses it, then the system should communicate it to the installer. If it isn't, then there will be quite difficult for the installer to find the origin of all the malfunctions of the system due to the installation error.

When a software system is installed in a certain environment, the installer will sometimes have to adapt the system to that environment. The "Configurability" of the system deals with the difficulty of configuring the system for the working environment (hardware, operating system, user…).

An installation process as succinct as possible is desirable, because it means less effort for the installer and, sometimes, less confusion. The "Succinctness" of the process bears on its "Shortness" and "Precision" (all the required things, and only these, will be installed). For example, during the installation of an operating system in an embedded system without display, once selected a system without display, the selection of the screen shouldn't appear. One thing very useful for shorting the installation process is the development of the "Autoconfigurability" (for example, Plug'n'Play techniques) of the system as much as possible.

The "Versatility" of the process is very important in order to facilitate the installation. The installer should be able to modify previously selected options (going back, for example) or to correct erroneous entries. If there are different ways to do things and different installers could use any of them, the installation process should provide them, especially in order to adapt the process to the installer level. For example, to select an installing path the user should be able to write it directly or to look for it in a tree structure.

The “Configurability” of the installation deals with the possibility of configuring the characteristics of the system that will be installed. For example, selection of the system path, selection of modules to be installed and so on.

In some cases the “Uninstallability” of the system is necessary, especially if the system is an application. Sometimes this is an implicit non-functional requirement in some systems. For example, who could imagine an application for Microsoft Windows without the possibility of uninstallation?

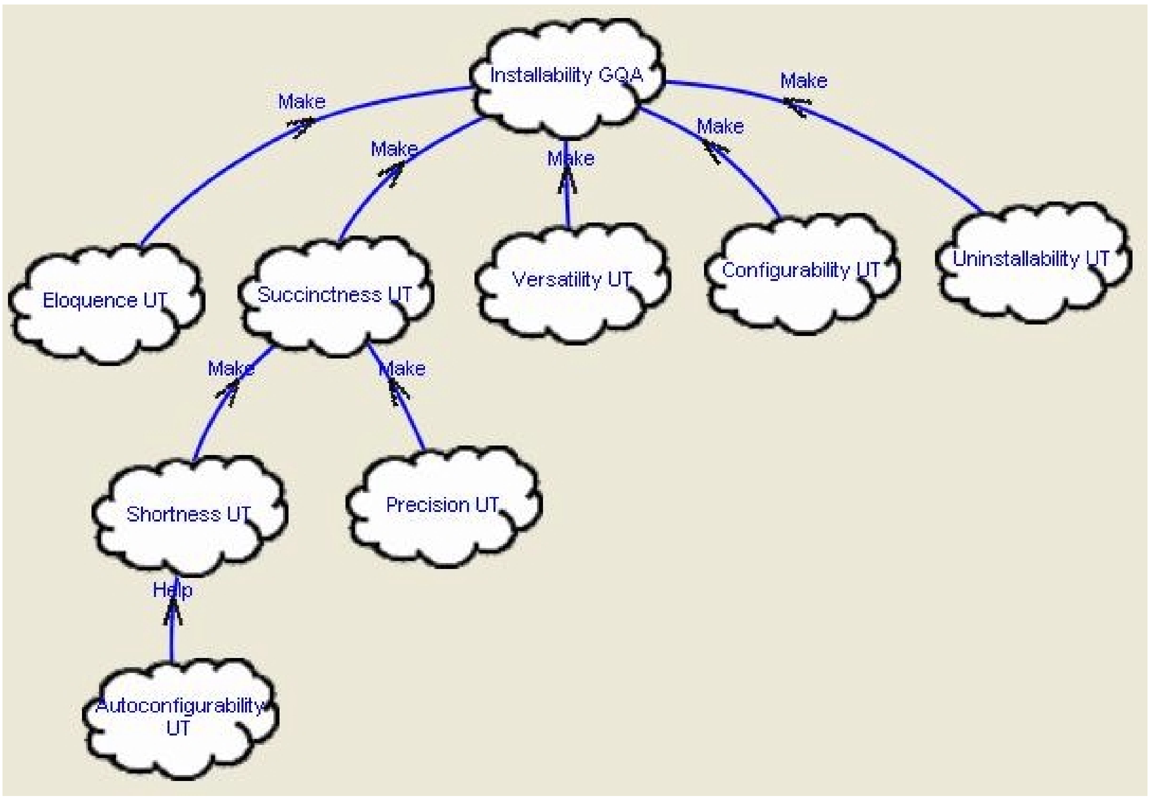


Figure 5-68: Installability decomposition

5.3.6.6.3 Replaceability

The name of this attribute can be confusing, but it isn’t if the ISO definition is carefully read: “opportunity and effort of using it in the place of specified other software in the environment of that software”. It has to be clear that it doesn’t mean that the software can be easily replaced. This attribute applies only to those systems that could have eventually to replace others.

The compatibility of a system could be divided in two different aspects: environment compatibility and interface compatibility. The first one is the compatibility of the system with the environment, in which it will be installed, what has been called replaceability. The second one is the compatibility of the system with the systems which it should be able to interact with, what has been called interoperability. Replaceability deals with the portability of the system and interoperability deals with the functionality of the system, so compatibility doesn’t exist as a characteristic or subcharacteristic and has been divided in these two subcharacteristics. Let’s see the decomposition of replaceability (see Figure 5-69).

The better the “Hardware compatibility” of a system is, the wider the range of systems that it can be able to replace will be. The hardware compatibility of a system, with other that will replace, is the capability of the system to manage the same hardware as the previous one without modifications. There are certain working environments in which some hardware devices are very common (for example, flash memories in embedded systems) and these devices should be almost necessarily supported by the system. A good example of systems with good hardware compatibility (in order to replace another system) is the applications for Microsoft Windows: if one application was already using a certain driver in order to manage a device, the most probably is that the driver is already present in the system.

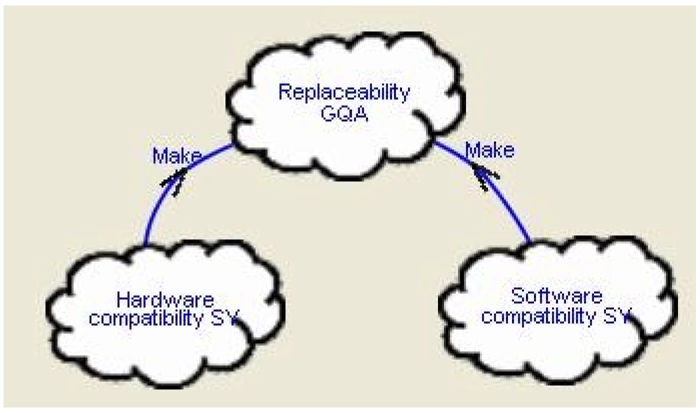


Figure 5-69: Replaceability decomposition

The studied system won’t be usually the only software in the complete system, thus it has to interact with other software systems. Therefore the “Software compatibility” of the system will be very important for the replaceability of the system. The software compatibility of a system, with other that will replace, is the capability of the software to run in the same software environment as the previous one without modifications. For example, the compatibility with the operating system (if there is one) is very important.

This environment compatibility (hardware and software compatibility) is also an aspect that improves the adaptability, installability and the co-existence ability of the system, but it is the most important aspect for the replaceability, therefore it is here explicit modelled and not in the other subcharacteristics.

Although this is a separate subcharacteristic, it involves other subcharacteristics of portability as adaptability and installability. Perhaps one system wasn’t thought to replace another system or systems; but if it’s adaptable, it will be able to replace other systems more easily. One step in the process of replacing a system with a new one is installing the new one. Thus the better the adaptability and the installability of a system are, the better its replaceability is.

If the software is an upgrade of previous software, the replaceability of the new software is very important for the final use of the system. The replacing should leave unchanged the functionality of the system or even improve it, but never get it worse. The replaceability of the new version of the system is an implicit non-functional requirement in every software upgrading.

5.3.6.6.4 Co-existence

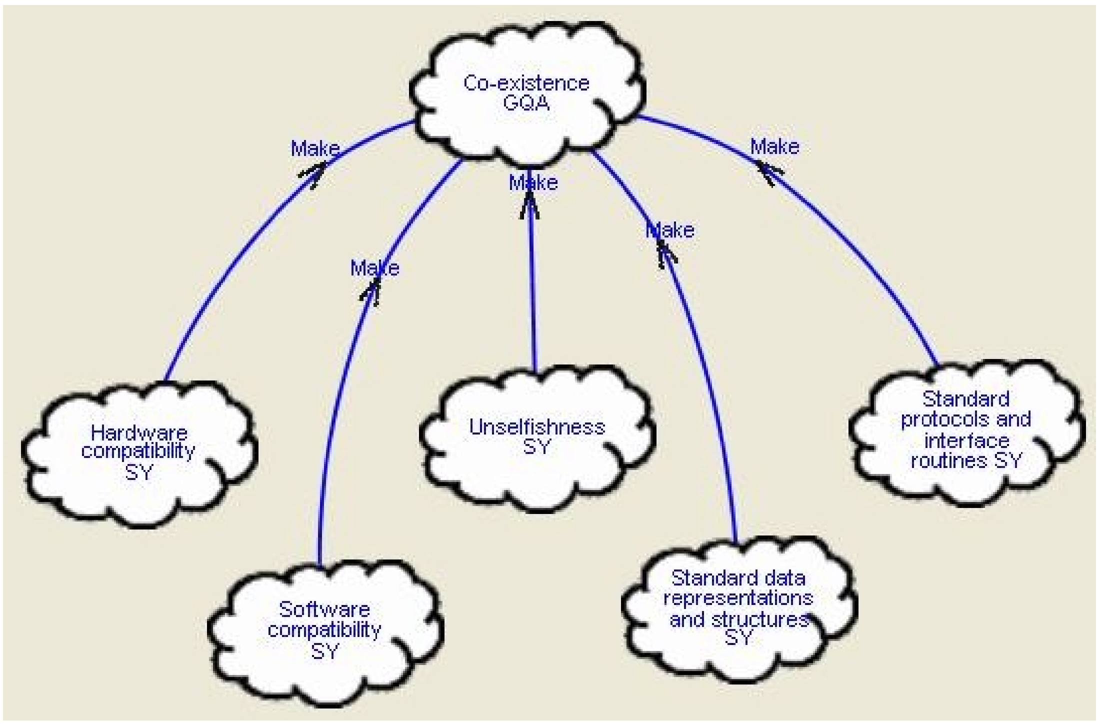

Co-existence GQA
Make
Make
Make
Make
Make
Hardware compatibility SY
Software compatibility SY
Unselfishness SY
Standard data representations and structures SY
Standard protocols and interface routines SY

Figure 5-70: Co-existence decomposition

The better the "hardware and software compatibility" of the system, the easier its co-existence with other systems is. For example, an application that supports Access and xBase databases will not have co-existence problems with another application that uses the same databases (but for other purposes) and only support Access databases.

The "Unselfishness" in the use of the resources that other software systems can try to share, if it is possible, is a good way of easing the co-existence with other systems. It is selfish the blocking of resources, but it is also selfish the excessive use of resources (more memory reserved than needed). For example, a selfish application that plays sound would block the soundcard for itself; an unselfish application would not block it.

The use of "Standard data representations and structures" and of "Standard protocols and interface routines" eases the sharing of data.

5.3.7 Conclusion and Future Work

In this chapter, we have presented an approach for eliciting, documenting and consolidating requirements. There are three major innovations. One is the use of a quality model and quality attribute types to capture general knowledge on NFRs, while specific NFRs are captured in a template. The second innovation is experience based approach, providing first quality models integrating quality attributes and means from customer and developer views. The third is the use of detailed checklists on how to elicit NFRs. With this approach, we achieve a minimal, complete and focused set of measurable and traceable NFRs.

We will investigate the following aspects in our future research:

- Improving the questionnaire : The questionnaire only addresses a selection of quality aspects at the moment (maintainability, efficiency, usability and reliability) and should in future be completed in order to cover the whole set described in ISO9126. In order to increase the validity of the questionnaire, the standardized statements should address different groups of stakeholders, such as developers or customers. Enlarging the questionnaire would on the one hand decrease the criteria economy but could on the other hand increase its applicability in order to not only determine the priority of the high level quality aspects, but to build a tailored quality model and identify low level attributes, that are needed in the further refinement.
- In the context of the Empress project, most work was done in the field of customer view QAs. More work has to be done on developer view QAs and the deployment of means in the checklists.
- How to ensure that the NFRs are well-formulated? The NFRs should be unambiguous, i.e., having only one interpretation, also for many stakeholders. We have to investigate which means can support a person which is writing down the NFRs, e.g., in a workshop. The tags of the NFRs shouldn’t be repeated and they should be unambiguous.

5.3.8 Appendix

5.3.8.1 Appendix A: Checklist for Efficiency:

The following checklist is an example for a checklist for efficiency. Depending on the project, the company and their understanding of quality (which is captured in the tailored quality model), a checklist can have different content. In text in italics gives the place to document the NFRs in the template.

1. **Organizational Experience** (*Project Issues->Process stakeholders section*)
   - Do you expect a certain experience in building time critical systems? If yes, specify the level of experience! (e.g., had a similar project before, has special qualification (training, certification)).
   - Do you expect a certain experience in building systems with resource limitations? If yes, specify the level of experience! (e.g., had a similar project before, has special qualification (training, certification))
2. **Missing efficiency NFRs**
   <u>If you used a default refinement tree</u>, check the following:
   With the help of the initialization, you specified basic efficiency requirements

based upon the functional requirements. Do you have any other efficiency NFRs that were not elicited during initialization (quality model did not cover all aspects)? If yes, specify them (*specify these efficiency requirements in a separate section)*

3. **Elicitation of boot / start up time requirements**
   For each system that has to be booted / started, specify the boot-time / startup-time of the HW and SW.
4. **Elicitation of response time / usage time NFRs** *(quality attributes->efficiency ->response time ->high level)*
   - Specify response time or usage time NFRs (depending on the UC-type) on the level of the UC-names (e.g., if the UC-name is "warn user", the NFR can be "In case machine A is running out of resources, the user has to be warned within 1 minute."
5. **Existing system constraints** (*separate section in requirements document or in related chapter (e.g., networking), usage of overview graphics where possible*)
   a. Are there any hardware-constraints on the specific parts of the physical system architecture? E.g., on number of processors, processor speed, memory of devices, networks between devices). Please note that only constraints and not requirements shall be specified at this point! If there are constraints, specify them.
   b. Are there constraints on the operating systems of the system/subsystem? If yes, which one? (e.g., Windows, Linux, Windows CE, Palm OS)
6. **Refinement of response time / usage time NFRs** *(quality attributes->efficiency ->response time ->detailed level)*
   For each UC that has a response time or usage time NFR:
   - Take the first UC-activity and try to express the response time / usage time NFR on this step.
   - Try to find the system architecture elements that are affected by this NFR (PDA, network, DB) and express the response time NFR on each of them.
7. **Elicitation of Throughput NFRs** *(quality attributes->efficiency ->throughput)*
   For each network in the system architecture:
   - Go through each Use Case: Think of an **average** usage of the Use Case regarding the following aspects:
     - Amount of data
     - Number of accesses on the element
     - Number of users

     Specify the throughput NFRs on the element of the physical system architecture.
   - Go through each Use Case: Think of a **maximum** usage of the Use Case regarding the following aspects:
     - Amount of data
     - Number of Accesses on the element

- Number of Users

Specify the throughput NFRs on the element of the physical system architecture.

8. **Elicitation of Workload Distribution NFRs** *(quality attributes->efficiency ->workload distribution)*
   This refinement does only make sense, if there are at least two devices per system architecture element (e.g., two processors). For each of these elements with at least two devices:
   - Go through each Use Case: Think of an **average** usage of the Use Case regarding the following aspects:
     - How shall the different instances of the Use Cases be distributed on the devices.
     - Within a Use Case: (How) shall the Use Case aactivities be distributed to the devices?

     Specify the workload distribution NFRs on the element of the physical system architecture.
   - Go through each Use Case: Think of an **maximum** usage of the Use Case regarding the following aspects:
     - How shall the different instances of the Use Cases be distributed on the devices.
     - Within a Use Case: (How) shall the Use Case activities be distributed to the devices?

     Specify the workload distribution NFRs on the element of the physical system architecture.
9. **Elicitation of Capacity NFRs** *(quality attributes->efficiency ->capacity)*
   For each element of the physical system architecture (see system architecture):
   - Go through each Use Case: Think of an **average** usage of the Use Case regarding the following aspects:
     - Which data has to be stored/processed?
     - Where will the data be stored/processed?

     Collect all this information and specify the capacity NFRs on the element of the physical system architecture.
   - Go through each Use Case: Think of a **maximum** usage of the Use Case regarding the following aspects:
     - Which (amount of) data has to be stored/processed?
     - Where will the data be stored/processed?

     Collect all this information and specify the capacity NFRs on the element of the physical system architecture.

5.3.8.2 Appendix B: Examples of Classifications

5.3.8.2.1 B.1: Classification of Failures

A classification of failures is useful for the contractor to specify reliability requirements. He could, for example distinguish the allowed percentage of faults by the class of failure:

- Cosmetic (user interface), e.g., a button is not shown 100%.
- Minor (wrong calculations), e.g., in a time planning system, during calculations a 1% offset appears.
- Major (subsystem crashes), e.g., a module of a program does not respond.
- Severe (total system crashes), e.g., the complete system does not respond.
- Gradual Failure (a failure slowly develops), e.g., memory leaks

5.3.8.2.2 B.2: Classification of Products

A classification of products is useful for the developer to see which products are most likely to change in future (useful for specifying maintainability requirements):

- Executable
- Code
- Documents:
    - Analysis Documents
    - Design Documents
- Benchmarking Results
- Test Cases
- User Manual (also visible to the Customer visible!)
- Technical Specification of HW (also visible to the Customer visible!)

5.3.8.3 Appendix C: The Priorization Questionnaire

This appendix consists of, first, the priorization questionnaire and, second, the instructions for interpreting the questionnaire.

Priorization Questionnaire

The purpose of this survey instrument is to get an impression of your project context. The results will be used to tailorize the nonfunctional requirements refinement method.

PART A: facts and experiences

INSTRUCTIONS: Consider each item separately and rate each item independently of all others. Circle the rating that indicates the extent to which you agree with each statement. Please do not skip any rating. If you do not know about a particular area, please circle N/A.

5 = Strongly Agree **4** = Generally Agree **3** = Neutral (acceptable)
2 = Generally Disagree **1** = Strongly Disagree **N/A** = Not Applicable

Please rate the following **facts** and **experience:**

1.	In case of changes to the system, we have to find the affected system parts.	5	4	3	2	1	N/A
2.	Our system or parts of it are working on limited hardware resources.	5	4	3	2	1	N/A
3.	Updates of the system are planned.	5	4	3	2	1	N/A
4.	If users don´t come along with the product, we might loose money.	5	4	3	2	1	N/A
5.	Main parts of the product will never change.	5	4	3	2	1	N/A
6.	There are laws/standards that regulate how (easily) the system has to be used.	5	4	3	2	1	N/A
7.	When modifying the system, it is difficult to ensure the validity of the system.	5	4	3	2	1	N/A
8.	The system uses less resources than available.	5	4	3	2	1	N/A
9.	After modifications, the system must be tested.	5	4	3	2	1	N/A
10.	For the success of the application, time is critical.	5	4	3	2	1	N/A
11.	There is fixed functionality, even in case of change it will stay the way it is.	5	4	3	2	1	N/A
12.	If the user doesn´t feel comfortable with the look of the software, he might stop using it.	5	4	3	2	1	N/A
13.	Understanding other peoples project documents is difficult in our company.	5	4	3	2	1	N/A

14. There are laws/standards that regulate the system's handling of faults, failures and how often it crashes.	5	4	3	2	1	N/A
15. There are laws/standards regulating the way, changes have to be made to the system.	5	4	3	2	1	N/A
16. Assuming the system has a few faults, they are not allowed to lead to complete system failure.	5	4	3	2	1	N/A
17. There are laws regulating the time behaviour and resource utilization of the product.	5	4	3	2	1	N/A

Comments ____
____
____
____

PART B: wishes

INSTRUCTIONS: Consider each item separately and rate each item independently of all others. Circle the rating that indicates the extent to which you agree with each statement. Please do not skip any rating. If you do not know about a particular area, please circle N/A.

5 = Very important **4** = Generally important **3** = Neutral (I don´t care)
2 = Generally unimportant **1** = Very unimportant **N/A** = Not Applicable

Please rate the importance of the following **wishes**:

18. Our systems shall be so easy to use, that trainings are not necessary.	5	4	3	2	1	N/A
19. The customer shall like the look and feel of the product.	5	4	3	2	1	N/A
20. Removing a feature from the product shall be easy.	5	4	3	2	1	N/A
21. The overall product shall remain stable, even if some parts fail.	5	4	3	2	1	N/A
22. The software shall enable the users to perform their tasks in the most intuitive way.	5	4	3	2	1	N/A
23. The time for a system to be down shall be exactly specified.	5	4	3	2	1	N/A
24. The first time I use the system, it shall enable me to judge if it supports me in my tasks.	5	4	3	2	1	N/A
25. The software shall use exactly the amount						

	of resources (memory, etc.) available.	5	4	3	2	1	N/A
26.	If the system crashes, it shall maintain basic functionality.	5	4	3	2	1	N/A
27.	Task A is one of my usual tasks. It shall be possible for me to fulfill this task with the program easily.	5	4	3	2	1	N/A
28.	Our software shall be 100% free of defects.	5	4	3	2	1	N/A
29.	Our company shall guide the user, so no mistakes can be made.	5	4	3	2	1	N/A
30.	Even if the system is threatened, it shall remain robust.	5	4	3	2	1	N/A

Comments

Instructions for interpreting the questionnaire

- Fill in the values checked for each question in the table below.
- Afterwards calculate the sum of the values and enter it in the sum field.
- In the next field enter the number of valid values.
- In the last field calculate the QA- score, according to the formula described in the table.
- To calculate the scores for every single sub-characteristic of maintainability, for each sub-characteristic in the table below, calculate the score by following the formula described in the same field.

Maintainability score:

A1	A13	A3	B3	A5	A11	A7	A9	A15	SUM= A1+A13+A3+ B3+A5+A11+ A7+A9+A15	Number of valid values (9-(N/A)-(missing values))	QA Score= SUM / Number of valid values

Analyzeability (A1+A13)/2	Changeability (A3+B3)/2	Stability (A5+A11)/2	Testability (A7+A9)/2	Compliance to maintainability standards: A15

Efficiency score:

A2	A10	A8	B8	A17	SUM= A2+A10+A8+B8+A17	Number of valid values	QA Score= SUM / Number of

						(5-(N/A)-(missing values))	valid values

Time behaviour (A2+A10)/2	Resource utilisation (A8+B8)/2	Compliance to efficiency standards: A17

Usability score:

A4	B1	A12	B2	B5	B7	B10	B12	A6	SUM= A4+B1+A12+ B2+B5+B7+ B10+B12+A6	Number of valid values (9-(N/A)-(missing values))	QA Score= SUM / Number of valid values

Learnability (A4+B1)/2	Attractiveness (A12+B2)/2	Understandability (B5+B7)/2	Operability (B10+B12)/2	Compliance to usability standards: A6

Reliability score:

A16	B11	B4	B13	B6	B9	A14	SUM= A16+B11+B4+ B13+B6+B9+ A14	Number of valid values (7-(N/A)-(missing values))	QA Score= SUM / Number of valid values

maturity (A16+B11)/2	Fault tolerance (B4+B13)/2	recoverability (B6+B9)/2	Compliance to reliability standards: A14

Priorization

- Now fill in the QAs and their scores into the fields of the following table and sort them (highest score gets rank 1):

Ranking order of the high level quality attributes		
Rank	Quality attribute	QA scores
1		
2		
3		
4		

- If priorization is necessary on the level of the subcharacteristics, fill their scores into the following table and sort them (highest score gets rank 1):

Ranking order of the sub-characteristics		
Rank	Sub-characteristics	Sub-characteristic scores

1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		

5.3.8.4 Appendix D: Example for Template

The following is an example of a requirements document template for the documentation of NFRs:

1 Overview

This chapter gives an overview of the document and collects all naming conventions and references. Furthermore, it allows the reader a quick overview of the purpose of the system to be developed, its function blocks, as well as the major constraints.

1.1 Purpose of the document

This section describes the authors and users of this document. For each author and user, it is described why he/she wrote/uses the document.

1.1.1 Authors of the document

This subsection describes the authors of this document and gives a brief motivation for it.

Authors

Motivation

1.1.2 Users of the document

This subsection describes the users of the document. For each user, the usage purpose is described.

-Contractor

-Subcontractor

1.2 Scope of the system

This section describes the scope of the system in terms of purpose, customers, users, function blocks, and main constraints. All requirements implied by this scope are listed explicitly in one of the sections: functional or non-functional requirements.

1.2.1 Purpose of the system

This subsection describes the purpose of the system.

1.2.2 Customers of the system

This subsection describes the customers of the system.

1.2.3 Function blocks

This subsection describes the function blocks of the system.

1.2.4 Main constraints

This subsection describes the main constraints of the system.

Organizational experience

1.3 Naming

This section describes conventions, abbreviations, and definitions for the whole project.

1.3.1 Conventions

This subsection includes syntactic conventions.

1.3.2 Abbreviations

This subsection includes agreed abbreviations.

1.3.3 Definitions

This subsection includes common definitions of terms. Monitored and controlled variables are defined in Sections 3.1 and 3.2.

1.4 References

This section refers to the following standards, company internal documents, and legal documents.

1.4.1 Standards

This subsection refers to standards that are relevant for the system.

1.4.2 Further documents

This subsection refers to company internal and legal documents that are relevant for the system.

1.5 Remainder overview

Chapter 2 „Context description“ describes the usage context, general constraints, assumptions, and dependencies of the system.

Chapter 3 “Variables” defines the monitored/input variables and the controlled/output variables of the system.

Chapter 4 “Functional requirements” describes the behavior of the system from the viewpoint of the user.

Chapter 5 “Non-functional requirements” describes constraints on the system, e.g., constraints derived from production or performance constraints.

Chapter 6 “Project issues” collects all issues relevant for project execution, namely plans, documentation, risks, open issues, off-the-shelf solutions.

2 Context description

This chapter supplies background on the application context of the system. This is only explanatory. All requirements implied by this context are listed explicitly in one of the sections: functional or non-functional requirements.

2.1 Usage context

This section describes constraints derived from the usage context (e.g., series, country, planned selling date).

Type Series

Countries

2.2 General constraints

This section describes general constraints on the system and the production process (e.g., numbers of production, sales, or domain constraints).

2.2.1 Production and process constraints

This subsection describes production and process constraints.

2.2.2 Domain constraints

This subsection describes domain constraints.

2.3 Assumptions and dependencies

This section describes assumptions and dependencies. This supports the check of their validity during project execution.

2.4 User characteristics

This section describes characteristics of the system's users that have to be considered during development.

3 Variables

This section describes all controlled and monitored variables that are mentioned in this document.

3.1 Monitored variables dictionary

This section describes all environmental variables monitored by the system

Name	Description

3.2 Controlled variables dictionary

This section describes all environmental variables that are controlled by the system.

Name	Description

4 Functional requirements

This section describes how users will use the system.

4.1 Overview of system

This section gives an overview on the function blocks of the system.

4.2 Use Cases

This section gives a set of use cases for each function block.

4.2.1 Group 1 of UCs

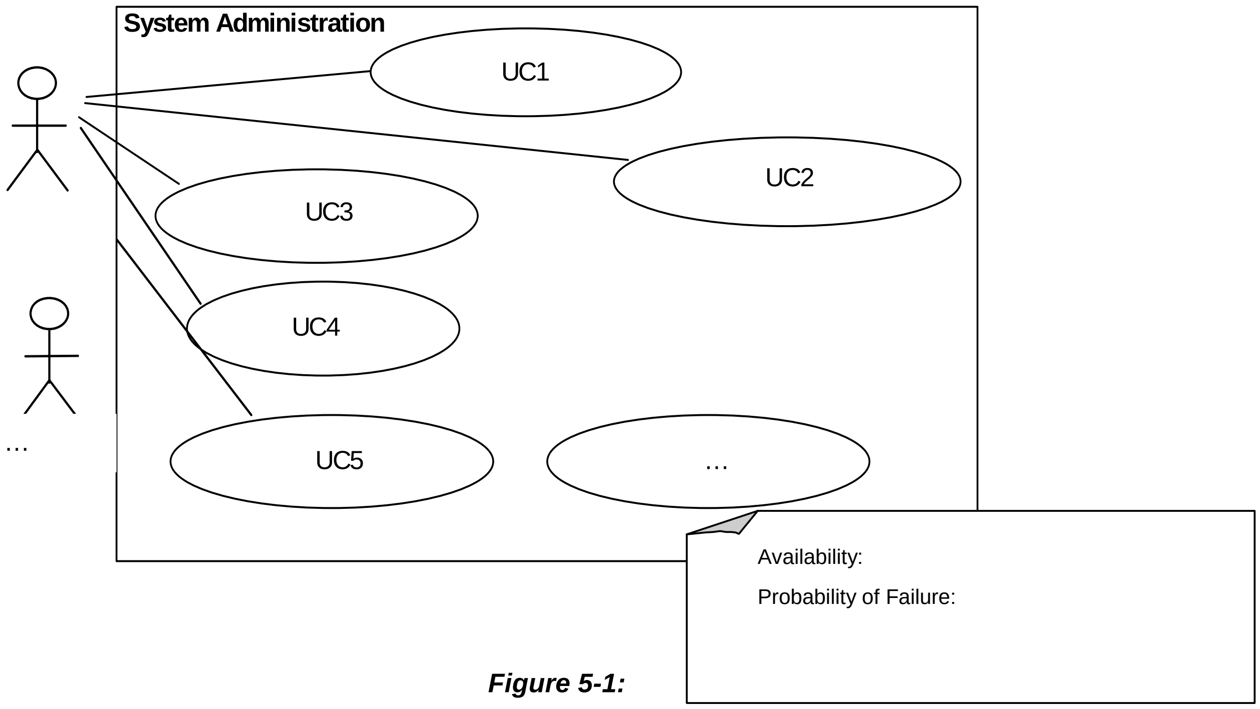


Figure 5-1:

UseCase	UC1
Actors	
Intent	
Preconditions	
Flow of events	
Exceptions	
Rules	
Quality constraints	
Monitored environmental variables	

Controlled environmental variables	
Postconditions	

UseCase	…
Actors	
Intent	
Preconditions	
Flow of events	
Exceptions	
Rules	
Quality constraints	
Monitored environmental variables	
Controlled environmental variables	
Postconditions	

4.2.2 Group 2 of UCs

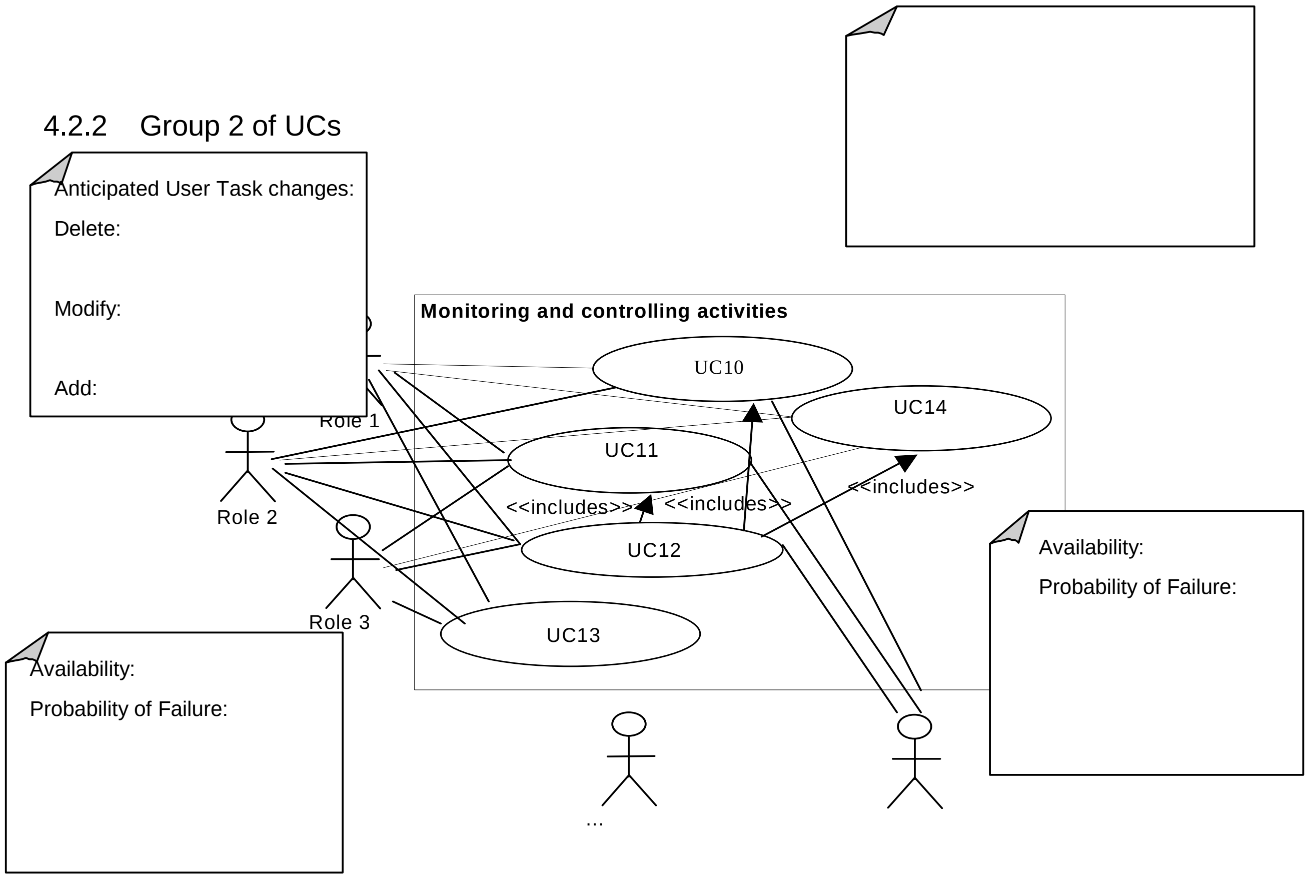


***Figure 5-2: Use Case-Diagram* "Monitoring and controlling activities"**

UseCase	UC10
Actors	
Intent	
Preconditions	
Flow of events	1. 2. 3.
Exceptions	1.1 2.1 2.2
Rules	
NFRs	**Maintainability:** Anticipated Changes to this task: Add: Modify: Delete: **Reliability:**

	Fault Tolerance: Downtime: **Efficiency:** Response / usage time: 1. 2. 3.
Monitored environmental variables	
Controlled environmental variables	
Postconditions	

5 Non-functional requirements

This chapter describes constraints on the system. This includes production, domain, operational, performance, and implementation constraints. In addition, this chapter describes assumptions and dependencies. If a human user interface is part of the system, also constraints on the user interface are given.

5.1 Production requirements

This section describes production constraints.

5.2 Domain requirements

This section describes constraints given by the domain.

5.3 Operational requirements

This section describes constraints that result from the operation of the system.

5.3.1 Selling market

This subsection describes constraints that result from the selling market of the system.

5.3.2 Electrical constraints

This subsection describes the electrical constraints of the system.

5.4 Quality requirements

This section describes NFRs of the system.

5.4.1 Constraints on overall System Architecture

5.4.2 Reliability

5.4.3 Maintainability

5.4.3.1 Anticipated Changes to System Architecture (Adaptive maintenance)

.

5.4.4 Efficiency

5.4.4.1 Throughput

Throughput shall be specified per system component:

System Component 1:

System Component 2:

5.4.4.2 Boot Time

The boot time shall be specified per system component:

System Component 1:

System Component 2:

5.4.4.3 Capacity

The capacity shall be specified per system component:

System Component 1:

System Component 2:

5.4.4.4 Workload

The workload shall be specified per system component:

System Component 1:

System Component 2:

5.5 Implementation requirements

This section describes implementation constraints.

5.6 Assumptions and dependencies

This section describes assumptions and dependencies. This supports the check of their validity during project execution.

5.7 User-Interface requirements

This section describes user-interface constraints.

6 Project issues

This section collects all issues relevant for project execution, namely plans, documentation, risks, open issues, and off-the-shelf solutions.

6.1 Process stakeholders

6.2 Process requirements

Efficiency:

Maintainability:

Reliability:

6.3 Documentation requirements

Efficiency:

Maintainability:

Reliability:

6.4 Time plan

This section gives a time plan for the development of the system.

6.5 Cost plan

This section gives a cost plan for the project.

For the complete project, the following costs are calculated:

Tool costs

Personal costs

Total costs:	Σ.

Maintenance-cost for corrective changes shall be 0€ for one year!

6.6 Prototyping requirements

This section describes requirements concerning the availability of prototypes.

6.7 Acceptance procedures

This section describes certain requirements concerning the acceptance procedure of the system.

6.8 Risks

This section lists possible project risks.

6.9 Open issues

This section describes issues that should be implemented in the next increment of the system and that have to be discussed in more detail.

6.10 Off-the-shelf solutions

5.4 Non-Functional Requirements Analysis for Component Exchange

5.4.1 Introduction

Evolvability is a non-functional requirement (NFR) of software systems, whose realization is a focus of the EMPRESS project. ISO/IEC 9126 (with some extensions from IEEE 1061) defines a Quality Model, which provides a classification of NFR. The class *Internal Quality Requirements* also contains the characteristics *maintainability* and *portability*, which have the sub-characteristics *changeability* and *replaceability*. Evolvability has a close relation to these requirements. Part of evolvability can be *exchangeability* of software components during runtime as well as during configuration time (bootstrap time) or design time.

The Analysis of (NFR) e.g. using a tree based framework implies structured decomposition of NFR into specific topics according to the system to be developed. The use of software patterns helps to analyze the influence of design approaches on the fulfillment of NFR. To specify the patterns for a specific system all possible design approaches and features have to be analyzed according to their technical requirements, their dependencies and their relation to NFR.

In order to analyze component exchange system features have to be considered in more detail. Such technical features and design approaches, like states or concurrency of components have a large influence on the character of component exchange and the fulfillment of other NFR like maintainability, portability, efficiency and usability.

Fraunhofer FIRST main focus in the EMPRESS project is reconfiguration during runtime. In the following chapter we discuss concrete component features and resulting consequences for the component exchange.

5.4.2 Analysis of Runtime Exchange

Dependent on the system design the component exchangeability can be defined during design time, configuration time and during runtime (reconfiguration). We concentrate here on runtime exchange because of the stronger influence of different technical features on the essential activities to perform the runtime exchange. A similar analysis with less restrictions could be executed for design-time and configuration-time exchange.

We chose three features, which have a strong impact in the exchange process:

1. Does the component hold a state? (Is it stateful?)
2. Is the component active or passive?
3. Is the component able to be accessed concurrently?

Components which only answer client-requests are referred to as passively. A component that acts independent of request handling is called active. Concurrent access, on the other hand, means that multiple concurrent execution threads can access the component at the same time.

To be able to concentrate on these three features we make the following assumptions to component system:

- In order to install a new component in the system during runtime the runtime system must be able to dynamically load binary code. An example is the like the class loader of Java.
- The runtime system has to support dynamic binding, to be able to bind dynamically loaded code to compiler-generated references.
- We assume a system architecture in which each component is accessed via an interface proxy, regardless whether the access the component is local or remote with respect to a client. This means that no direct calls to interface functions are allowed. This requirement drastically simplifies the exchange of components. However the overhead for undocking the component from a running system is increased.
- Interfaces are defined by contracts that include not only signatures but also the semantics and quality of service aspects. That is, there can be no such thing as a "changing interface".

Based on these requirements we establish the following list of essential activities and consequences, which must be taken into consideration.

1. The state of the component has to be consistent.

   Before Exchange, it has to be ensured that a stateful component reaches a consistent state so that no data or state information get lost during the exchange.

2. No thread is within the component.

   In case of a multi-threaded system, no thread executes the component. All running client-requests are processed when the exchange starts.

3. Client-requests have to be handled during component exchange.

   Multi-threaded systems have to deal with incoming request during component exchange. Two possible ways are request queuing and global request stop signalization.

4. Internal threads have to be stop.

   Active Components have to finish their internal activities.

5. Perform component exchange including state transfer (if states exists).

6. Disable forwarding of client-requests during state transfer.

Stateful components with concurrent access have to deal with requests during state transfer.

The following table shows the lists of the relevant points for every possible combination of the three considered features.

Has state?	Is active?	Concurrent access?	Action
False	False	False	5
False	False	True	2,3,5
False	True	False	4,5
False	True	True	2,3,4,5
True	False	False	1,5
True	False	True	1,2,3,5,6
True	True	False	1,4,5
True	True	True	1,2,3,4,5,6

Table 1: Resulting activities for different requirement combinations

With the help of the information in this table it is possible to build software patterns, which can be used in the NFR-framework to analyze a given system. For the development of a particular system more than the described features have to be considered to investigate the influence of the exchangeability on other non-functional requirements.

6 Appendix

6.1 Literature

[ACD93] Alur, R., Courcoubetis, C., and Dill, D.L., 1993, Model checking in dense real-time, Information and Computation, 104(1):2(34).

[AES01] Álvarez, J., Evans, A., and Sammut, P., 2001, MML and the Metamodel Architecture, http://www.2uworks.org/documents.html.

[Atk01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. Component-based Product Line Engineering with UML. Component Software Series. Addison-Wesley, 2001.

[AutoFOCUS] free case tool available from http://autofocus.in.tum.de

[AW01] Ivan Araujo and Michael Weiss, “Patterns and Non-functional Requirements: An interim Report”, Carleton university, 2001

[AZ02] I. Alexander and T.Zink, “An Introduction to Systems Engineering with Use Cases”, paper submitted to CCEJ, 2002

[Bau00] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In Proceedings of ICRE-2000, 2000

[Bay99a] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR’99), Los Angeles, CA, USA, May 1999. ACM.

[Bay99b] J. Bayer, D. Muthig, and T. Widen. Customizable Domain Analysis. In Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE ’99), Erfurt, Germany, Sept. 1999.

[Ber] E. V. Berard, “Be Careful With “Use Cases”, The Object Agency, Inc. http://www.toa.com/pub/html/use_case.html

[Bid02] R. Biddle, J. Noble, and E. Tempero. Supporting Reusable Use Cases. In Proceedings of the Seventh International Conference on Software Reuse, Apr. 2002.

[BLM02a] Vieri Del Bianco, Luigi Lavazza, Marco Mauri “A Formalization of UML Statecharts for Real-Time Software Modeling”, The Sixth Biennial World Conference on Integrated Design Process Technology (IDPT 2002), Pasadena, California, 23-28 June 2002.

[BLM02b] Vieri Del Bianco, Luigi Lavazza, Marco Mauri “Model Checking UML Specifications of Real-Time Software”, The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland, 2-4 December 2002.

[Boe78] Boehm et al.,“Characteristics of software quality”, 1978

[BS00] Manfred Broy, Oscar Slotosch: From Requirements to Validated Embedded Systems. EMSOFT 2001: 51-65. 2000

[Cam03] http://dictionary.cambridge.org/

[CDK01] G. Chastek, P. Donohoe, K.C. Kang, S. Thiel, “Product Line Analysis: A Practical Introduction, Technical Report, CMU/SEI-2001-TR-001, June 2001

[CEK01] Clark, T., Evans, A., and Kent, S., 2001, Engineering modelling languages: A precise meta-modelling approach, http://www.2uworks.org/documents.html.

[Che76] Chen P.P., “The entity-relationship model – towards a unified view of data”, ACM Transactions on Database Systems, 1(1):9-36, 1976

[Cha01] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel. Product Line Analysis: A Practical Introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, June 2001.

[Chu00] E.Chung et al, “Non-Functional Requirements in Software Development”,. Kluwer Academic Publisher, 2000

[Cle01] P. C. Clements and L. Northrop. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, Aug. 2001

[Coc01] A. Cockburn. Writing Effective Use Cases. Addison Wesley, 2001.

[Com97] comp.object “A Critique of Use Cases”, July 1997, http://ootips.org/use-cases-critique.html

[CS] Empress Case Study from Siemens: “CMWE: Control Monitor and Wireless Extension“(this is a preliminary version that will evolve in future!)

[Cza00] K. Czarnecki and U. Eisenecker. Generative programming: methods, tools and applications. Addison-Wesley, 2000.

[D1.1] Empress deliverable D1.1: “Requirements Engineering focusing on non-functional requirements”

[D3.2.2] Empress deliverable D3.2.2: “Method, notation, process for management of requirements in product families and across products”

[Dei01] B. Deifel, “Requirements Engineering komplexer Standardsoftware”, Dissertation, TU München, München, 2001.

[Den02] Christian Denger High Quality Requirements Specifications for Embedded Systems through Authoring Rules and Language Patterns, Diploma Thesis, University Kaiserslautern, 2002

[Doors] Telelogic DOORS Homepage
http://www.telelogic.com

[Dou98] B. P. Douglass Real-Time UML, Addison Wesley, 1998.

[DP01] Dutoit, A.H., B. Paech, P., “Rationale Management in Software Engineering”.In: S.K.Chang (Ed.), “Handbook of Software Engineering and Knowledge Engineering. World Scientific, December 2001

[DS] Quasar project: “Document structure of requirements documents”

[Fow99] Martin Fowler, „UML Distilled. A Brief Guide to the Standard Modelling

Language.", Addison Wesley, 1999

[Gam94] Gamma et al, "Design Patterns", Adisson Wesley, 1994

[GBJ00] M. Glinz, S. Berner, S. Joos, J. Reyser, N. Schett, R. Schmid, Y. Xia, "The ADORA Approach to Object-Oriented Modelling of Software", Research Report 2000.07, Institut fur Informatik, University of Zurich. [vedere se pubblicato]

[GGJ00] C.A. Gunter, E.L. Gunter, M. Jackson, P. Zave, "A Reference Model for Requirements and Specifications", IEEE Software, vol. 17, n. 3, May-June 2000.

[Gil02] T. Gilb, "Practical Advanced Requirements Engineering“, slideset, http://www.result-planning.com, June 2002

[Gli00] Martin Glinz, "Problems and Deficiencies of UML as a Requirements Specification Language", 10th International Workshop on Software Specification and Design, San Diego, Nov. 2000, p.11-22.

[Gom00] H. Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, Proceedings of the Sixth International Conference on Software Reuse, June 2000.

[Gri98] M. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In Proceedings of the Fifth International Conference on Software Reuse, Vancouver, BC, Canada, June 1998.

[Hat88] D.J. Hatley; I.A. Pirbhai: Strategies for Real Time System Specification. New York, Dorset House Publishing 1988.

[Hou01] Frank Houdek, "Beispiel-Systemspezifikation: Türsteuergerät", DaimlerChrysler AG, Forschung und Technologie, 2001

[Hur97] R.R. Hurlbut "A Survey of Approaches For Describing and Formalizing Use Cases", http://www.iit.edu/~rhurlbut/xpt-tr-97-03.html

[IEEE98] IEEE-Std 830-1998 IEEE Guide to Software Requirements Specifications, The Institute of Electrical and Electronics Engineers, New York, 1998

[ISO01] ISO/IEC 9126-1:2001(E), "Software Engineering-Product Quality-Part 1: Quality Model",2001

[Jac01] M. Jackson, "Problem Frames - analysing and structuring software development problems", Addison-Wesley ACM Press, 2001.

[Jac95] M. Jackson. Software requirements and specifications. Addison-Wesley, 1995.

[Jac87] I. Jacobson, "Object-Oriented Development In an Industrial Environment," OOPSLA '87 Conference Proceedings, special issue of SIGPLAN Notices, Vol. 22, No. 12, December 1987, pp. 183 - 191.

[Jac92] I. Jacobson. Object-Oriented Software Engineering, A USE Case Driven Approach. Addison Wesley, 1992.

[Jac97] I. Jacobson, M. Griss, and P. Jonsson. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, 1997.

[JoDo03a] I. John, J. Dörr. Extracting Product Line Model Elements from User Documentation. Technical Report, Fraunhofer IESE, 2003, to appear

[JoDo03b]	I. John, J. Dörr Elicitation of Requirements from User Documentation. Proceedings of REFSQ '03, Klagenfurt, June 2003
[Joh01]	Isabel John. Integrating Legacy Documentation Assets into a Product Line. In Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, October 2001.
[Joh02a]	Isabel John and Dirk Muthig Tailoring Use Cases for Product Line Modeling, REPL Workshop , Essen , September 2002.
[Joh02b]	Isabel John and Dirk Muthig Product Line Modeling with Generic Use Cases, Workshop on Techniques for Exploiting Commonality Through Variability Management, SPLC 2, San Diego, August 2002.
[Kan90]	K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.
[Kne02]	Antje von Knethen Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems. PhD Theses in Experimental Software Engineering, Vol. 9, Fraunhofer IRB, 2002
[Kol02]	Elena Kolodizki, "From Non-Functional Requirements to Design through Patterns", 2002
[Kor98]	Timothy Korson, "The Misuse of Use Cases", Object Magazine, May 1998, http://www.korson-mcgregor.com/publications/korson/Korson9803om.htm
[KP92]	Kesten, Y., Pnueli, A.: Timed and Hybrid Statecharts and their Textual Representation. In: Vytopil, J. (ed.): Proc. of FTRTFT92. LNCS, Vol.571. Springer-Verlag, Berlin Heidelberg New York (1992).
[Kru00]	Philippe Kruchten, "The Rational Unified Process – An Introduction", Addison Wesley, 2000
[Laue03]	Soren Lauesen, "Task Descriptions as Functional Requirements", IEEE Software, March/April 03.
[Loo99]	E.E. Loos Glossary of linguistic terms; published on CD- ROM by SIL International, 1999
[LQV01]	Luigi Lavazza, Gabriele Quaroni, Matteo Venturelli "Combining UML and formal notations for modelling real-time systems", Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), Wien, 10-14, September 2001.
[McC94]	McCall`s definition of "Quality factors" (pp. 959-969) in "Encyclopedia of software engineering", Vol.2, Marciniak 1994
[meta]	http://www.metamodel.com visited 13.6.2003
[Mey88]	Bertrand Meyer, "Object-Oriented Software Construction", Prentice Hall International, 1988.
[Mut02]	Dirk Muthig: A Lightweight Approach Facilitation the Incremental Transition Into Software Product Lines . PhD Theses in Experimental Software Engineering, Fraunhofer IRB, to appear, 2002
[NaC]	Northrop and Clemens: Software Product Lines, Addison Wesley, SEI series

[Schm02] Klaus Schmid: A Comprehensive Product Line Scoping Approach and Its Validation. In Proceedings of the 24th International Conference on Software Engineering (ICSE'02), 2002

[SGW99] B. Selic, G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modeling, Wiley, 1999.

[SMC74] W.G. Stevens, G.J. Mayers and L.L. Constantine, "Structured Design", IBM Systems Journal, vol.13, n.2, p.115-139, 1974.

[Som01] I. Sommerville Software Engineering, Addison Wesley 2001

[SSV98] Sommerville, I., Sawyer, P. und Viller, S.: "Viewpoints for requirements elicitation: a practical approach, Proceedings of ICRE", 1998

[Ste98] R. Stevens, P. Brook, K. Jackson, and S. Arnold. Systems Engineering – Coping with Complexity. Prentice Hall, 1998.

[SR01] Kendall Scott and Doug Rosenberg "Applied Use Case Driven Object Modeling", Addison-Wesley, 2001. [verificare]

[TUMaster] B. Bajraktari: Modelbasiertes Requirements Tracing. Master thesis, TU München, 2001

[OMG01] Object Management Group. OMG Unified Modeling Language Specification, Version 1.4, September 2001.

[OMG02] OMG, UML Profile for Schedulability, Performance, and Time Specification, Final Adopted Specification, March 2002.

[ODM96] Software Technology for Adaptable, Reliable Systems (STARS). Organization Domain Modeling (ODM) Guidebook, Version 2.0, June 1996

[Par98] H. Partsch : Requirements Engineering Systematisch. Universität Ulm, Springer Verlag 1998.

[PM95] D.L. Parnas and J. Madey, "Functional documentation for computer systems", Science of Computer Programming, 25:41-61, October 1995.

[Robe99] S. Robertson, J. Robertson. Mastering the Requirements Process Addison-Wesley, 1999

[RJB99] Rumbaugh J., Jacobson I., Booch G., "The Unified Modelling Language Reference Manual", Addison-Wesley, 1999

[UWG] The precise UML group, http://www.cs.york.ac.uk/puml.

[W3.2] Empress Deliverable W3.2: "Method for tracing non-functional requirements"

[whatis] www.whatis.com ; http://whatis.techtarget.com/definition/0,,sid9_gci211982,00.html visited 13.6.2003

[Wie03] Karl E. Wiegers, "Software Requirements", Microsoft Press, 2003

[WSE] DaimlerChrysler Case Study: "Wash System Example", ?

[Wei99] D. M. Weiss and C. Lai. Software Product Line Engineering: A Family Based Software Development Process. Addison-Wesley, 1999.

[Wie99] K. E. Wiegers, "Software Requirements", Redmont, Microsoft Press, 1999

[YU?]	E.Yu, “From Non-Functional Requirements to Design through Patterns”, University of Toronto.
[2UC]	2U Consortium Unambiguous UML, http://www.2uworks.org/documents.html.
[2UC01a]	Initial submission to ad/00-09-02 (UML 2.0 Superstructure), 22/10/2001, http://www.2uworks.org/documents.html.
[2UC01b]	Initial submission to ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL), 22/10/2001, http://www.2uworks.org/documents.html.