Certified Professional for Requirements Engineering

Handbook

Requirements Modeling

Practitioner | Specialist

Thorsten Cziharz
Peter Hruschka
Stefan Queins
Thorsten Weyer

2.2.0 | July 23, 2024

Terms of Use

Acknowledgements

Our thanks to Torsten Bandyszak, Sibylle Becker, Nelufar Ulfat-Bunyadi, Ruth Rossi,

Tracy Duffy, and Stefan Sturm for their support in the preparation of the manuscript.

This Handbook was produced by (in alphabetical order):

Thorsten Cziharz, Dr. Peter Hruschka, Dr. Stefan Queins, and Dr. Thorsten Weyer

The compilation of this Handbook was supported by

PALUNO
The Ruhr Institute for Software Technology

SOPHIST

Translated from German by:

Ed van Akkeren, Lars Baumann, Jan Jaap Cannegieter, Colin Hood, Peter Hruschka, Matthias Lampe, Ellen Leutbecher, Hans van Loenhoud, Piet de Roo, Stefan Staal, and Johan Zandhuis

Foreword

This *Handbook* complements the syllabus of the CPRE Requirements Elicitation module.

This Handbook is intended for training providers who want to offer seminars or training on the CPRE Requirements Elicitation Practitioner and/or Specialist according to the IREB standard. It is also aimed at training participants and interested parties who want to get a detailed insight into the content of this module.

This Handbook is not a substitute for training on the topic. The Handbook represents a link between the Syllabus (which lists and explains the learning objectives of the module) and the broad range of literature that has been published on the topic.

The contents of this Handbook, together with references to more detailed literature, support training providers in preparing training participants for the certification exam. This Handbook provides training participants and interested parties an opportunity to deepen their knowledge of Requirements Engineering in an agile environment and to supplement the detailed content based on the literature recommendations. In addition, this Handbook can be used to refresh existing knowledge about the various topics of requirements elicitation, for instance after having received the Requirements Elicitation Practitioner or Specialist certificate.

Suggestions for improvements and corrections are always welcome!

E-mail contact: info@ireb.org

We hope that you enjoy studying this Handbook and you will successfully pass the certification exam for the IREB CPRE Requirements Modeling Practitioner or Specialist.

More information on the IREB CPRE Requirements Elicitation can be found at: http://www.ireb.org.

Version history

Version	Date	Comment
1.1	September 2015	First release of the English version of the Handbook based on the original German version (1.0). Contains some minor changes compared to the original German version v1.0.
1.2	May 2016	Minor bugfixing and language polishing.
1.3	August 2016	Content on the topic "modeling of association classes" added and minor corrections.
2.0.0	July 2022	Inconsistencies between German and English version fixed. In detail: ▪ Chapter 3.5.3.1 misspelling and numeration figure 1,2,3 in 23, 24, 25 ▪ Chapter 3.5.3.2 inserted ▪ Chapter 3.7.1 text insertet and a new reference ▪ Chapter 4.3 paragraph inserted ▪ Chapter 4.3.2.1 text inserted ▪ Chapter 4.3.7 numeration figure 47 in 50 ▪ Chapter 4.4.4.6.1 text fixed ▪ Formatting ▪ Disclaimer: Gender-sensitive text formulation ▪ Addition to the references ▪ Inclusion of the Advanced Level split in Practitioner and Specialist
2.1.0	May 2024	New Corporate Design implemented, term "Advanced Level" eliminated, update terms of use and foreword
2.2.0	July 2024	Figure 43 complemented by further the main model elements of activity diagrams (terminator, object node, pin, signal transmitter, event receiver, time event). Chapter 4.3.7 wrong term "timer event" event fixed.

Table of Contents

IREB CPRE module Requirements Modeling

In recent years, the scope and complexity of typical software-based systems have increased significantly. This is reflected directly in the number of requirements arising and the complexity in terms of the mutual dependencies between requirements. All forecasts about the expected future increase in the size and complexity of software-based systems predict that the number of requirements and the complexity of interdependencies will continue to increase dramatically in the future. This becomes clear, for example, if we consider the development trends in the field of business information systems in terms of the Internet of Services (IoS) and Internet of Things (IoT) or the development in the field of intelligent embedded systems. Both trends are paving the way for a somewhat revolutionary penetration of the physical world by dynamic networked software-based systems, referred to as "cyber-physical systems".

The first thing to note is that requirements are taking a central role in the development process of software-based systems. What is more, the extent and complexity of the requirements of a system are becoming more difficult to handle. Accordingly, the specification of requirements has already reached its limits in many areas if this is done only in natural language (i.e., in text form). In many cases, this has a lasting negative effect on the development projects concerned. Due to the many advantages of using graphical models with respect to readability, controlling complexity, automatic analyzability, and the processing of extensive and complex situations, the use of graphical modeling of requirements is increasing rapidly.

The IREB Certified Professional for Requirements Engineering module Requirements Modeling provides the tools for specifying requirements of large and complex systems using standardized and widely used modeling languages. Comprehensive tool support is available for these modeling languages—from freeware tools to powerful commercial CASE tools, there is great potential for automation and for seamless integration with other tools used in development processes (e.g., for project and test management).

More information on the IREB Certified Professional for Requirements Engineering module Requirements Modeling can be found at: http://www.ireb.org.

1 Basic principles

Requirements play a fundamental role in the life cycle of systems. In particular, the Fehler! Textmarke nicht definiert.development disciplines (such as architecture, design, implementation, and testing) are based mainly on the requirements of the system as specified during requirements engineering and are largely dependent on the quality of these requirements. In addition to the development disciplines, activities such as maintenance and service right up to decommissioning of the system and development of upstream activities (e.g., assessment of the risks and costs of the development project) depend highly on the requirements and their quality.

According to the *IREB Glossary of Requirements Engineering Terminology* [Glin2011], a requirement is (1) a need that is perceived by a stakeholder or (2) a capability or property that a system must have. Requirements engineering is concerned with ensuring that the requirements of the system under development are formulated as completely, correctly, and precisely as possible, thereby providing optimal support for the other development disciplines and activities in the life cycle of the system.

1.1 The benefits of modeling requirements

Using a highly simplified example, Figure 1shows the difference between textual and modeled requirements. The left-hand side shows four textual requirements which specify necessary behavior in relation to the input of data via an entry screen. The right-hand side shows a requirements diagram in which the corresponding requirements are modeled.

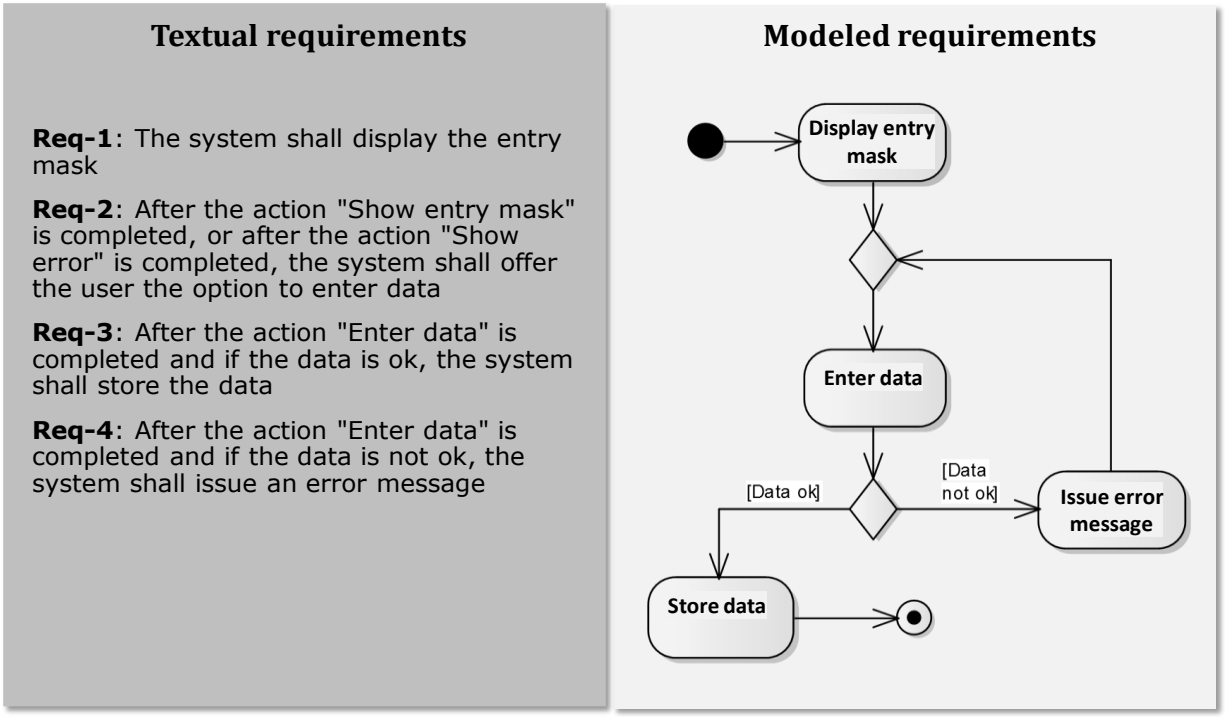


Figure 1: Textual requirements vs. modeled requirements

As this simple example already indicates, modeling the requirements shows the necessary behavior of the system in a more structured and understandable way. The reader can follow the process step by step. Furthermore, this simple example clearly shows that the interaction of the various aspects of the required system behavior are explicitly visible in the modeled requirements, whereas this information is only implicitly present in the textual requirements (see also [Davi1993]).

Typically, software systems today comprise significantly more complex processes, meaning that the associated textual requirements are very extensive and complex. It is then difficult for the reader to understand the interactions within such complex processes solely on the basis of textual requirements.

1.2 Applications of requirements modeling

Today, there are various applications for modeling requirements in requirements engineering, as explained in this section:

1.2.1.1 Modeling requirements as a means of specification

In this case, requirements diagrams replace textually specified requirements. This means that requirements diagrams are used as the primary means for specifying the system requirements or part of the system requirements. The requirements diagrams can (and should) be supplemented by textual requirements or textual explanations, specifically when a text is more compact or easier to handle than diagrams.

If all requirements still need to be available in textual form (e.g., due to contractual conditions or certification requirements), they can be generated from the requirements models—for example, using templates for converting requirements diagrams into text form.

1.2.1.2 Modeling existing textual requirements for the purpose of testing

In this case, a requirements diagram is created for a logically coherent set of textually specified requirements which, for example, specify a necessarily complex system behavior. The purpose of this diagram is to check the comprehensibility of textual requirements or to uncover inconsistencies or omissions in the textual requirements. Any defects uncovered are then corrected in the textual requirements.

1.2.1.3 Modeling existing textual requirements for clarity

In this case, for example, modeled requirements are used to represent extensive and complex relationships that affect the behavior of the system. However, this redundant form of the specification can lead to significant problems with regard to contradictions between textually specified requirements and modeled requirements.

1.3 Terms and concepts in requirements modeling

Using the general terms and concepts found in system modeling, the following explanation looks at the terms and concepts relevant for modeling requirements as well as the important relationships between the various terms and concepts. 2 shows a semantic network of the basic terms and concepts relevant for requirements modeling. Terms that are already defined in the *IREB Glossary of Requirements Engineering Terminology* are labeled with ↑.

The system of terms is based on various definitions in the *IREB Glossary of Requirements Engineering Terminology* [Glin2011] and complements this glossary with terms and concepts that are particularly essential for requirements modeling. A *model* is regarded as an abstracting image of the properties of a *system*.

To make the scope and complexity of the modeling manageable, various *views* of the *system* (and its environment) and the properties of the system in relation to each specific view are represented through *diagrams* and supplementary *textual* model elements. Each diagram is based on a specific *diagram type,* which in turn is defined via a *modeling language* (more precisely by *syntax, semantics, and pragmatics)*. The underlying *modeling language* of a *diagram type* defines the set of *modeling constructs* that can be used to construct the corresponding *diagrams* (e.g., class and association for the construction of class diagrams). In a modeling language, graphical and/or textual notations are defined for the modeling constructs.

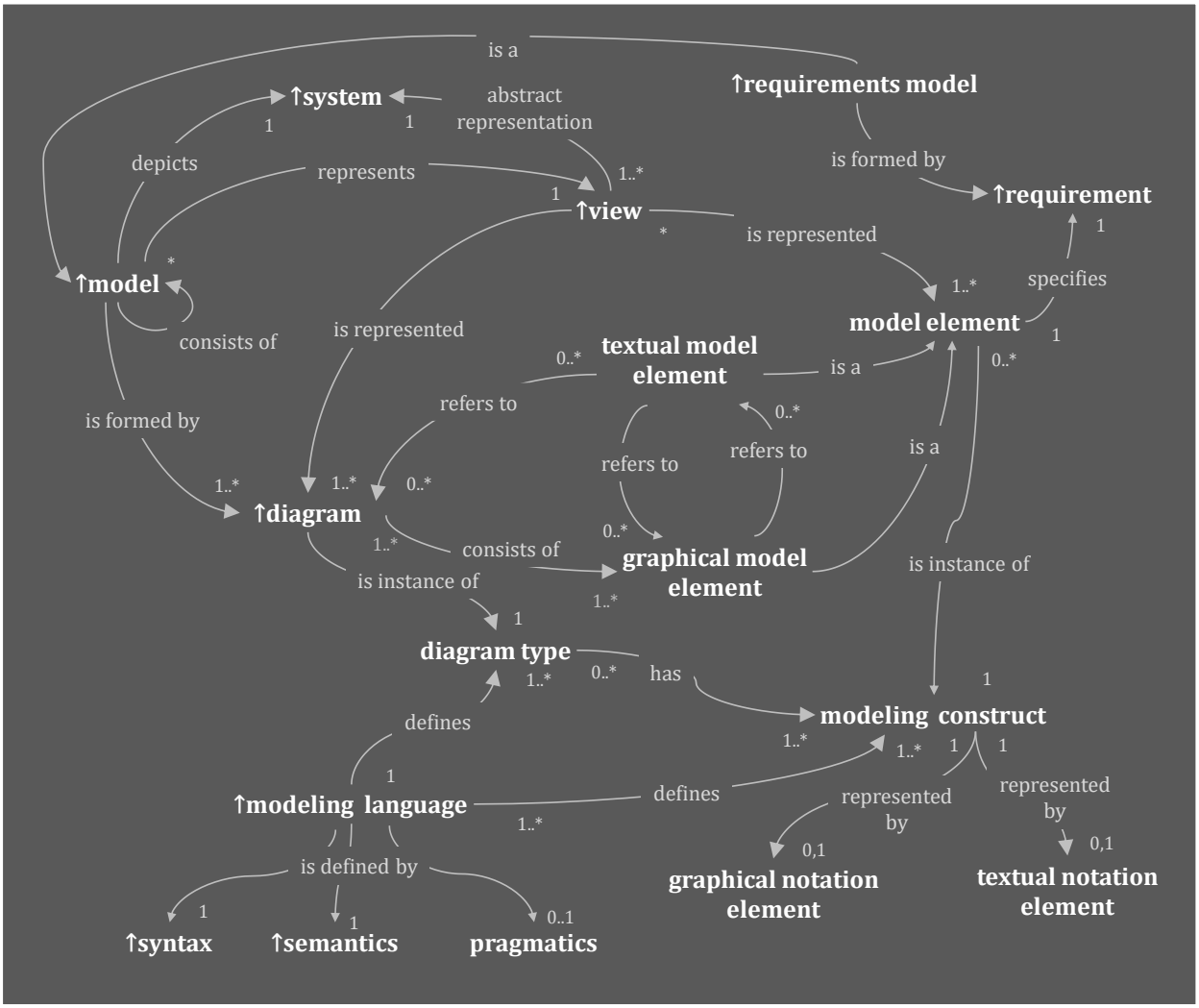


Figure 2: Conceptual network of the core terminology in requirements modeling

A diagram consists of a set of *model elements*, each representing a specific *graphical modeling construct* of the *modeling language* of the associated *diagram type* (e.g., class: "person", association: "is employed by", class: "company").

Diagrams and graphical model elements can be supplemented by *textual* model elements (e.g., textual description of the trigger of a use case) which express specific *textual modeling constructs* (e.g., a section of a use case template). The graphical and textual *model elements* form the atomic constituents of *models*.

A *requirements model* is a specific type of model (more precisely: a type of system model) used to specify the *requirements* of a system with the aid of *diagrams* and textual supplements.

1.4 Requirements models

The individual requirements of a requirements model are represented by model elements that are specified within requirements diagrams and via textual additions to these diagrams.

1.4.1 Modeling languages for requirements modeling

A number of diagram types and associated modeling languages are available for requirements modeling. The selection of the diagram type to be used in each case depends on the purpose, which thus determines which specific requirements of the system should be documented and which persons are the "target audience" for the requirements models.

The relevance of a diagram type often also depends on the type of system (e.g., operational information system or embedded system) and partly on the application domain (e.g., banks, insurance companies, automation technology, vehicle/aircraft industry) for which the system is being developed. Often (e.g., in embedded systems), requirements engineering focuses on the reactive behavior of the system. This is because the size and complexity of the required behavior of today's embedded systems are mainly determined by the necessary reactivity of the systems. Therefore, state machine diagrams of the OMG SysML [OMG2010a], OMG UML [OMG2010b], or MATLAB/Simulink Stateflow diagrams are used for requirements modeling when developing embedded systems. The state machine diagrams can be supplemented by complementary diagrams, such as use case diagrams, scenarios, or activity diagrams. In contrast, business information systems (e.g., software for processing loan applications) usually have no extensive and complex reactive behavior.

Therefore, when modeling requirements for such systems today, it is primarily diagram types that allow the modeling of extensive and complex information structures (e.g., UML class diagrams) that are used. Other diagram types used are those that allow the modeling of process-oriented aspects, such as event-based process chains [[Sche2000]] or BPMN diagrams [OMG2011] as part of the business analysis, as well as UML activity diagrams—for example, to model requirements with reference to the required flow logic of the system under development. Here again, other complementary types of diagrams can be used—for example state machine diagrams—in order to model the necessary requirements in terms of reactivity of the system.

In addition to specific approaches such as event-driven process chains (EPCs) or BPMN, which are often used in the context of business analysis or MATLAB/Simulink diagrams in requirements modeling for embedded systems, the "universal" modeling approaches UML and SysML are very often used for modeling requirements.

UML version 2.4 distinguishes between 14 different diagram types, seven of which are used for structure modeling and seven diagram types are used for behavior modeling. Note that the diagram type "profile diagram" is used to document language profiles (i.e., adaptations and extensions to the modeling language) and not, like the other diagram types, for actual system modeling.

SysML was designed specifically for modeling in the development of complex systems and is a subset of UML extended with special diagram types and notation elements.

The corresponding extensions relate to new structure diagrams (internal block diagrams, block definition diagrams, parametric diagrams). SysML no longer contains the diagram type "class diagram". With regard to the behavior diagrams, no new diagram types are introduced in SysML; instead, the behavioral diagram types of UML are used, whereby SysML activity diagrams differ from the UML activity diagrams with respect to syntax and semantics.

1.4.2 Requirements modeling versus system design

In practice, it is sometimes difficult to distinguish between requirements diagrams and design diagrams (see, e.g., [BoRJ2005]). The cause is frequently seen in the fact that the same universal modeling languages are used for requirements modeling, such as UML or SysML. In fact, the cause in most cases is that the alleged requirements diagrams specify not requirements but rather the system design, or that requirements and design are mixed in diagrams.

The latter is the case, for example, when the required system behavior is already modeled in relation to individual, specific design decisions in a diagram and these design decisions are not specified by boundary conditions (constraints), for example, in terms of the technology to be used (see Section 1.5).

1.4.2.1 Requirements diagrams and design diagrams in system analysis

As part of the system analysis, it is often the case that both design diagrams and requirements diagrams are created. The first step in system analysis is typically the analysis of an existing system. The "system" can be anything from an individual software system to complex socio-technical systems where a variety of software systems and people (or roles) cooperate in order to fulfill an overarching purpose, as is the case, for example, in complex business information systems.

The system analysis itself can be performed from different perspectives, such as function-centered or data-centered (see, e.g., [DeMa1979] and [ShMe1988]). In the context of system analysis, the system under development is often initially analyzed (e.g., the system in operation and the associated documentation) and modeled in the form of diagrams as it is perceived. In this case, the technical incarnation of the system is modeled first, that is, the concrete technical solution as it is in operation (see [McPa1984]).

The corresponding model of the incarnation is then analyzed in terms of the underlying technical aspects, meaning that it is abstracted from the concrete technical implementation to identify the business core. The result of this activity is a model of the functional requirements of the system under development.

Both models—the incarnation model (i.e., the technical solution) and the model of the functional requirements (also referred to as the essence model)—are factual models, that is, models that document the existing properties of the system under development (SuD). As part of the system analysis, a target model is then often formulated based on the model of the functional requirements.

This target model specifies which technical requirements are to be implemented by a newly developed system or as part of a change project. These technical requirements are then incorporated back into the development process. In typical systems analysis processes, therefore, both requirements diagrams and design diagrams are created. The goal of system analysis is to model the functional requirements of the system under development.

1.4.2.2 Relationship between requirements models and design models

During the development of complex software systems, requirements and design are often developed with very strong links. This close link between the development of requirements and the definition of a solution in the form of a system design is illustrated with the *twin peaks model* shown in Figure 3 (cf. [Nuse2001]).

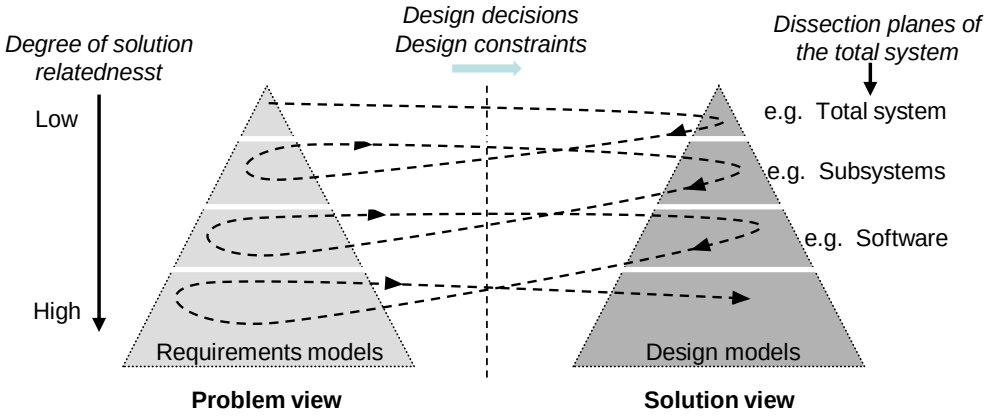


Figure 3: Relationship between requirements and design

As illustrated in the figure, during the development of complex software systems, there is a strong interaction between the definition of requirements and the system design. Typically, the first step is to produce a set of more general requirements for the complete system. This set of requirements is then the basis for the definition of the preliminary system architecture which satisfies these requirements.

During the transition between requirements definition and system design, design decisions have to be made and the given conditions for the design (design constraints) have to be met (e.g., the specification of a style of architecture to be used). Starting from the initial system architecture, which consists for example of (logical) subsystems, the requirements for the individual subsystems can be specified. If sufficiently detailed requirements are available, the initial system design is refined.

As an example, Figure 3 illustrates the relationship between the requirements and design of a technical system (complete system) which is initially abstracted from the separation between hardware and software. The requirements for the actual software of the system are first specified on the third system level.

For pure software development projects, the software to be developed is classified at the highest system level. On the lower system levels, *logical components* and *software parts* are then considered (see, e.g., [ISO26702], [HaHP2001]).

In this approach, the design decisions at one level significantly affect the definition of requirements at the next lower level of detail—that is, the requirements of the next level are based on the design decisions previously made which in turn represent a framework for the specification of requirements at the next lower level. Even though there is a close link between requirements and architectural design, within the scope of requirements modeling it is all the more important to strictly separate the requirements model from the design model and to establish the relationships through appropriate dependency relationships (see Section 1.9). More details can be found in [Pohl2010], [BDH2012], and [HaHP2001].

1.5 Views in requirements modeling

The foundation level of the Certified Professional for Requirements Engineering distinguishes between three views in the modeling of functional requirements (cf. [PoRu2011]), namely:

1. the static–structural view
2. the behavioral view
3. the functional view

Building on these basic views of requirements modeling, a more differentiating set of views is presented below (see Figure 4).[1]

[1] The creation of views can be established in various ways within the scope of requirements engineering. For example, views can be defined that address specific concerns of stakeholders. A "user view" can be defined of the requirements of the system, for example. This view considers (models) only those requirements that directly concern the use of the system under development. In a "maintenance engineering view", only those system requirements that relate directly to the maintenance of the system would be considered. Various "philosophies" for establishing views can be applied in combination to control the scope and complexity of requirements modeling. It is conceivable, for example, that the user view and the maintenance engineering view are each considered from an information structure view and a dynamic view. Through common concepts or mapping relationships, the requirements models of the different views can then be integrated into an overall model.

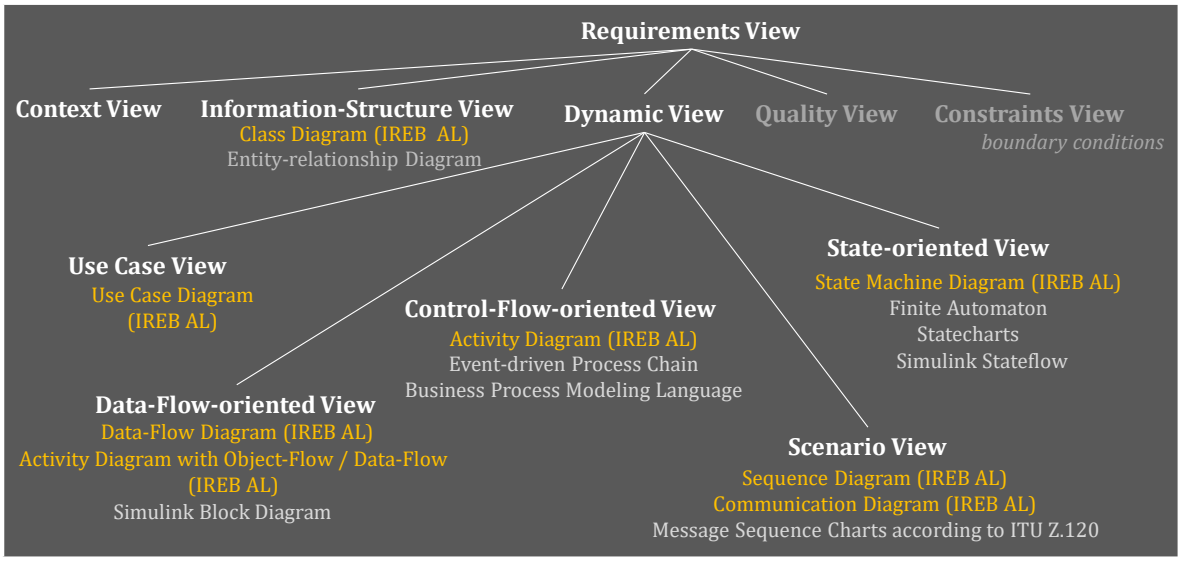


Figure 4: Views in requirements modeling in the IREB module Requirements Modeling

A key challenge in requirements engineering is to understand the context of the system under development (e.g., the software to be developed). This includes the knowledge of what other systems are related to the system under development in an operational context, properties of these external systems, as well as knowledge about which roles, people interact with the system and which properties they have that are relevant for the system.

Context modeling is typically used to identify the necessary interfaces between the system under development and its context.

1.5.1 Information structure view

The information structure view focuses on requirements of the system under development which are related to static and structural aspects of the functionality, such as the structure of data to be processed by the system. Typical diagram types used here are class diagrams or various dialects of entity–relationship diagrams (e.g., according to Chen or in the FMC approach).

1.5.2 Dynamic view

The dynamic view focuses on those requirements of the system under development which are related to dynamic aspects of the functionality (see, e.g., [BoRJ2005]). For the purposes of the foundation level of the Certified Professional for Requirements Engineering, the dynamic view of the requirements of a system is formed through the behavioral and functional views.

To model the requirements in the dynamic view, in Requirements Modeling, the dynamic view is strongly differentiated (see Section 1.6). Typical diagram types used for requirements modeling here are use case diagrams, activity diagrams, state machine diagrams, data flow diagrams, and sequence diagrams.

1.5.3 Quality view

The quality view focuses on those requirements of the system which relate to necessary qualities of the system under development or individual system components. Although there are a number of approaches for model-based specification of quality requirements currently being researched (see, e.g., [HKDW2012]), in practice today quality requirements (regarding, for example, performance, reliability, real-time behavior, safety, or robustness) are still specified within requirements models mainly by textual supplements or as an annotation to specific model elements in requirements diagrams (see, e.g., [RiWe2007]).

A detailed taxonomy of requirements in the quality view (quality requirements) can be found in ISO 25010 [ISO25010]. Detailed information on the documentation of requirements in the quality view can be found in [Pohl2010].

1.5.4 Constraints view

The constraints view focuses on requirements in terms of boundary conditions (i.e., external constraints) to be adhered to by the system under development (or the associated development process) (see [ISO29148]). Typical boundary conditions include organizational, regulatory, or technological conditions.

Technological constraints occur, for example, in the form of design constraints (e.g., service-based or client-server) which define a specific architectural style for the system under development.

Such constraints are often documented in textual form (or by textual additions in requirements models), whereas specific types of diagrams such as class diagrams or component diagrams are often also suitable for documenting organizational or technical constraints. Detailed information about boundary conditions can be found in [RoRo2006], for example.

1.6 Views of the dynamic view in requirements modeling

The dynamic view in requirements modeling considers those requirements which relate to the chronological-logical relationships in the required behavior of the system. Today's business information systems—and intelligent embedded systems even more so—have a very extensive and complex structure of such relationships. These relationships have to be elicited and analyzed and specified in the requirements as part of requirements engineering.

To make the scope and complexity of such dynamic relationships in the system behavior manageable within requirements modeling, the dynamic view is divided into views.

The integration of these views leads to an overall model of the dynamic view of the requirements of the system under development, as shown in Figure 4.

1.6.1 Use case view (user functions and dependencies to the system context)

Within the dynamic view, the use case view considers the high-level system user functions and their relationships to actors in the system context. A high-level user function characterizes a functionality that the system must offer for an actor within the context to gain a benefit (added value). Use case diagrams are typically used for modeling here.

1.6.2 Data flow-oriented view (system functions and data dependencies)

Within the dynamic view, the data flow-oriented view considers the functions that are perceptible at the system interface, as well as the data dependencies between these functions and with actors in the system context. The functions can also be analyzed at various levels of granularity, for example, from high-level user functions (e.g., use cases) to finely detailed technical functions, the interaction of which implements the functionality of the use case. Typical diagrams used here are data flow diagrams (e.g., according to DeMarco [DeMa1979]) and activity diagrams that focus on the object flow between actions.

1.6.3 Control flow-oriented view (process flow logic)

Within the dynamic view, the control flow-oriented view considers the processes (or activities or actions) perceptible at the interface of the system and their flow logic. The control flow relationships are considered in processes that occur, for example, in the form of sequential, alternating, or concurrent sequences.

UML or SysML activity diagrams are typically used to model the control flow-oriented view. A special feature with regard to business analysis is that (extended) event-driven process chains or BPMN diagrams are also used for modeling at business process level.

1.6.4 State-oriented view (states and state changes)

The required state space of the system is modeled in the state-oriented view within the dynamic view. In particular, the model shows the reactive behavior of the system in relation to the system context. The states and state changes that are observable at the interface between the system and the system context are modeled in this view. A state change of the system under development can be triggered by an event in the system context, by a time event, or by an intrinsic event.

Finite automata, Harel Statecharts, or UML state machine diagrams based on these concepts are typically used here.

1.6.5 Scenario view (interaction sequences between actors and the system)

The scenario view within the dynamic view considers interactions between actors in the system context and the system which lead to one or more actors in the system context obtaining added value or achieving a goal (e.g., obtaining cash by using an automated teller machine). Scenarios are frequently used to make use cases in use case diagrams more specific.

Here, the scenarios describe the interactions between the system and actors in the system context that lead to successful execution of the use case. In scenario modeling, as well as the immediate interaction between actors and the system under development, the message exchange between actors in the context of the system is also typically modeled. UML/SysML sequence diagrams or Message Sequence Charts according to the ITU standard Z.120 [ITU2004] are typically used to model scenarios.

1.7 Adapting modeling languages for requirements modeling

UML and SysML have a concept for adapting or extending the different modeling languages. This is useful, for example, when specific concepts of a project or application domain are to be anchored in the language. UML and SysML are typically adapted by defining stereotypes to give notation elements a special meaning (or semantics).

In UML and SysML, all notation elements can be adapted or extended by stereotypes. The definition of a stereotype consists of a syntactic part, in which the representation of stereotypes and the desired references to notation elements are set, as well as a semantic part which specifies the meaning of the stereotype.

In UML/SysML diagrams, stereotypes are modeled in the form of angle brackets. For example, using the stereotype << domain >> for classes within a class diagram (☒ definition of the syntax of the stereotype), it would be possible to express that classes that have this stereotype are specific to the particular application domain and their technical meaning is more precisely defined within a domain glossary (☒ definition of the semantics of the stereotype).

1.8 Integrating textual requirements in the requirements model

SysML differs from UML in that it has a special means of notation for modeling textual requirements. It also defines a special type of diagram, the *requirements diagram*, which is assigned to neither the structure view nor to the behavior view. This diagram type allows the modeling of relationships between textual requirements or the attachment of textually specified requirements to model elements of SysML diagrams and referencing of these requirements.

This type of "modeling" of textual requirements is often used to include predetermined requirements (e.g., from the point of view of a special field) in the requirements model.

The main purpose of this integration is to relate the modeled requirements to the predetermined textual requirements. This allows the expression of which modeled requirements make a textual requirement more specific.

Most commercially available UML tools, however, already offer the possibility of using textual requirements in any diagram type, and not only in *requirements diagrams*. This allows, for example, the specification of textual requirements as an alternative to the diagrammatic specification because in the opinion of a requirements engineer, certain requirements can be specified more appropriately in textual form. For example, an action in a flow can be refined through a number of textual requirements which are then included in the requirements model and related to this action (by means of an appropriate tracing relationship, for example).

Using this *concept of integrating textually specified requirements* in requirements models allows us to specify quality requirements that relate to a specific action (e.g., requirements concerning the performance of this action) as textual requirements by placing them in a relationship with the action within the diagram in which the action was modeled.

Through this *concept of complementary use of textual requirements*, model elements from the various diagram types for requirements modeling (and thus the corresponding diagrams) can be extended in order to relate textual requirements to requirements diagrams within a requirements model.

1.9 Documenting dependencies between model elements

Regardless of whether requirements are available in the form of requirements diagrams or in textual form, they can be linked to one another in the course of model-based documentation of requirements with UML/SysML using explicitly defined dependency relationships. To do this, appropriate stereotypes for dependency relationships between model elements of the requirements model can be defined (see also Section 1.7).

In many cases, the stereotype to be used (i.e., its syntax and semantics) depends heavily on the project context and the application domain, which means that in a development project, the project participants must define which dependency types are needed between requirements (see also [RaJa2001]). The required dependency relationships must then be defined in the appropriate tools.

Typical examples of commonly found dependency relationships between model elements within a requirements model are:

- **<<refines>>:** A <<refines>> B expresses that a single requirement or a set of requirements A refines a single requirement or set of requirements B by, for example, specifying one or more additional requirements to the requirements B.
- **<<realizes >>:** A << realizes >> B expresses that the requirements A realize the requirements B. This is used, for example, when A represents the requirements for a component that when met, lead to fulfilment of the requirements B for the entire system.

However, this type of tracing is based on the fact that either (1) design decisions about the structure of the solution were taken in the development process, or (2) the need for such a component or specifications about the structuring of the overall system into components already exist as boundary conditions for requirements engineering (cf. [BDH2012], for example).

- **<<satisfies>>:** A <<satisfies>> B expresses that a single requirement or set of requirements A meets a single or a set of requirements B. This type of dependency relationship is used, for example, in customer–supplier relationships when more detailed requirements that have been specified by the contractor have to be related to the more general requirements of the client to express that the requirements A of the contractor meet the requirements B of the client.
  This type of dependency is used to express relationships between requirements in the system requirements specification and requirements in the customer requirements specification—for example, to support evidence that, for the system under development, the requirements specified in the system requirements specification ensure that the realized system will meet the requirements in the customer requirements specification.
  The dependency type <<satisfies>> has a certain resemblance to the dependency type <<realizes>>, whereby dependencies of the type <<satisfies>> are typically used at the interface between client and contractor.

1.10 The benefits of requirements modeling

Compared to the textual specification of requirements, specification of requirements by means of diagrams has a number of essential advantages:

- **Requirements are easier to understand:**
  Cognitive research has shown that, generally, facts that are visualized in diagrams are easier to understand and remember than corresponding textual descriptions of these facts (cf. [LaSi1987]). In particular, this means that requirements specified in diagram form are easier to understand and remember than requirements which exist in textual form. "A picture is worth a thousand words!"
- **Inherent support of the principle of "separation of concerns":**
  Diagram types are defined for a specific purpose and, through the available notation elements (semantics) and the way the language allows these notation elements to be combined (syntax), force the modeler to focus on a situation. For example, state machine diagrams should be used to model the necessary reactive behavior of the system under development as part of requirements modeling and not to model processes or information structures. In requirements modeling, the separation of concerns is established by different views. The requirements models of the individual views can be integrated through common concepts. This allows us to make statements across different views of requirements. Detailed information can be found in [DaTW2012].

- **Inherent support of the principle "divide and rule":**
  By using different diagram types, the specific requirements supported by that particular diagram type can initially be modeled in isolation. The diagrams of different types can be combined using common concepts or defined mapping relations in order to obtain an integrated requirements model. This feature of diagram-based specification of requirements supports the requirements engineer in breaking down the overall problem— that is, the specification of the requirements of a system—into manageable sub-problems (e.g., the specification of requirements for a subsystem). The merging of the individual requirements models of the sub-problems then forms the requirements model of the higher level system. More detailed information can be found in [BDH2012] and [HaHP2001], for example.
- **Reduced risk of ambiguity:**
  Due to the higher degree of formality of modeling languages for requirements modeling compared to natural languages, requirements specified in diagram form have a lower risk of ambiguity or misinterpretation by other participants in the development process (e.g., the architects, developers, testers).
- **Higher potential for automated analysis of requirements:**
  Due to the higher degree of formality of requirements specified in diagram form compared to requirements specified in text form, such requirements can be analyzed to a large extent or even completely by machine (e.g., an analysis of the accessibility of states in a requirements diagram of the state-oriented view).
- **Higher potential for automatic processing of requirements:**
  The higher degree of formalization of requirements specified in diagram form also increases the possibility of processing the requirements of the system further automatically and using them in other development disciplines, for example, to derive test cases for system testing from requirements diagrams of the control flow-oriented view.
- **Requirements in context:**
  The modeling of requirements leads to individual model elements within the requirements model (see Section 1.3) and the relationships of individual requirements to other requirements being represented directly in the requirements model. This facilitates the handling of large and complex requirements and promotes understanding of the requirements because the context of a requirement is visible to the reader of the requirements in the requirements model. In an activity diagram, for example, for every action it is immediately visible what other actions this action is related to and what change of state of the system under development is triggered by the execution of the action.

1.11 The quality of requirements models

The quality of a requirements model is based on the quality of its components. As described in Section 1.1, the requirements model of a system is composed of a set of diagrams and textual additions. When requirements are modeled, a substantial part of the requirements is specified in the diagrams, which means that the quality of the requirements model is largely determined by the quality of the individual diagrams and their mutual relationships.

In turn, the quality of the individual diagrams is determined by the quality of the model elements within the diagrams and the associated textual additions. The left-hand pane in Figure 5 illustrates the hierarchical structure of the evaluation of the quality of requirements models.

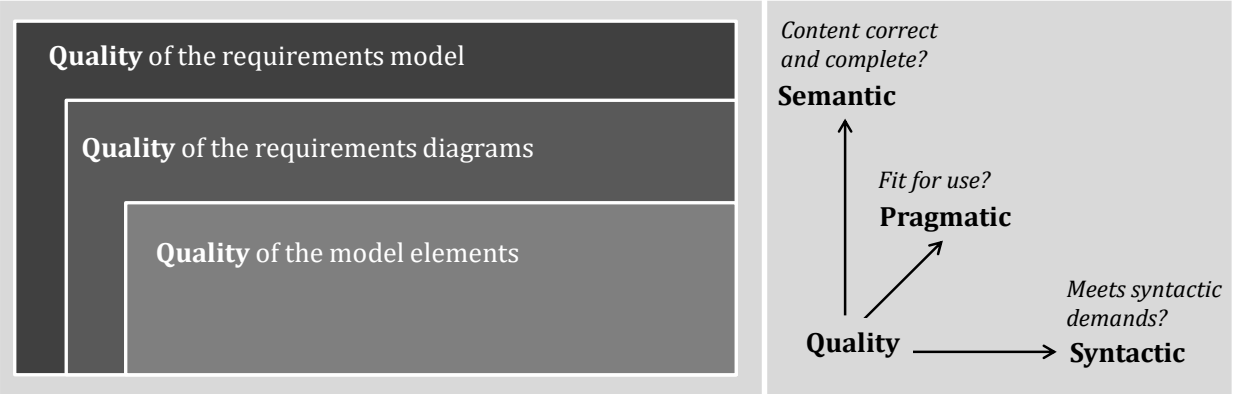


Figure 5: Assessment of the quality of requirements models

The quality of the requirements model, the requirements diagrams, and model elements can be assessed against three criteria (see [LiSS1997], for example):

- **Syntactic quality**
  The syntactic quality expresses the extent to which a single model element (graphical or textual), requirements diagram, or requirements model satisfies the applicable syntactic specifications.
  If the syntactic quality of a requirements diagram of the scenario view (which is in the form of a UML sequence diagram) is to be assessed, the extent to which this diagram meets the syntactic requirements of UML must be examined. For example, the syntax of sequence diagrams prescribes that a synchronous message at a certain level of detail consists of a function call and a reply message.
  If, in a scenario modeled by a sequence diagram, a reply message occurs without a preceding function call, this does not meet the syntactic specifications of the underlying modeling language and thus reduces the syntactic quality of the diagram.
  If appropriate modeling tools are used for modeling requirements, the syntactic quality of the diagrams created is usually ensured by the tool.

- **Semantic quality**
  The semantic quality expresses the extent to which a single model element (graphical or textual), the requirements diagram, or the requirements model correctly and completely represents the facts. Let us assume, for example, that after the insertion of a debit card into the card slot of an ATM, the customer's PIN is required as the first step.
  If a relevant requirements diagram of the control flow-oriented view (e.g., an activity diagram) models that after reading the card data, the customer is first asked for the payment amount, this represents a semantic defect in the corresponding diagram since the actual flow required deviates from the diagram.
  Such a defect in a requirements diagram negatively affects the semantic quality of the higher level requirements model.
- **Pragmatic quality**
  The pragmatic quality expresses the extent to which a single model element (graphical or textual), the requirements diagram, or the requirements model is suitable for the intended use. This in particular raises the question of whether the degree of detail and abstraction level is appropriate for the intended use. For a single model element, this means whether the model element (such as a state transition in a state-oriented requirements model) is specified at the right level of detail (e.g., is only the triggering event specified?
  Or are the additional conditions applicable for the state change and the triggered behavior indicated?). The pragmatic quality of an individual model element, a requirements diagram, or a requirements model can only be assessed if the addressee and the purpose of the diagram are known. Since the pragmatics determine what abstractions are useful, this also has a direct impact on the assessment of the semantic quality—that is, the completeness of a model element, a requirements diagram, or a requirements model can only be assessed in terms of an abstraction that is sensible from a pragmatic point of view.

1.12 Further reading

Terminology in requirements modeling

- Glinz, M.: Glossary of Requirements Engineering Terminology. Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version 1.1, May 2011.

Requirements modeling

- Pohl, K.: Requirements Engineering – Fundaments, Principles, Techniques. Springer 2010.
- Booch, G: Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley 2005.
- Daun, M.;Tenbergen, B.; Weyer, T.: Requirements Viewpoint. In: Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M.: Model-Based Engineering of Embedded Systems, Springer, Heidelberg 2012.
- Davis, A. M.: Software Requirements – Objects, Functions, States. 2nd Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

Quality of requirements models

- Lindland, O. I.: Sindre, G.; Sølverg, A.: Understanding Quality in Conceptual Modeling. IEEE Software, Vol. 22, No. 2, IEEE Press, 1994, 42–49.
- Pohl, K.: Requirements Engineering – Fundaments, Principles, Techniques. Springer, 2010.

2 Context modeling

A major challenge in requirements engineering is understanding the context of the system. The more complex and critical the system under development is, the more important it is to understand and document the context. This includes knowledge about which other systems influence the system under development in an operational context, properties of these external systems, as well as knowledge about which roles or persons interact with the system in an operational context and which properties that are relevant for the system they have. In addition, context modeling also helps to identify the necessary interface of the system under development.

2.1 Purpose

In requirements engineering, the scope of the system under development is defined (that is, the system boundaries are specified) and the system under development is clearly distinguished from its context. For this purpose, the influence of the context has to be investigated and ideally documented. The more complex and more critical the system under development is, the more important it is to document the knowledge about the context effectively. This includes the knowledge about:

- Which roles and persons interact with the system in operation?
- What other systems are related to the system under development from an operational perspective?
- How the interface between the system under development and the people and systems is created in context?

Furthermore, the context view can help when considering the properties (functions, qualities) of the external systems relevant for the system under development.

The context view documents properties of the system context. In contrast, the following chapters mainly specify the perceivable necessary properties of the system that are in scope and the system must have to fulfil its purpose in operation (including meeting the goals of stakeholders and thereby complying with all conditions). The context view thus documents a significant aspect of the work of requirements engineers when defining the interface between the system and the context.

2.2 Context diagrams

From a requirements perspective, the context view defines the scope of a system, meaning that it draws a line between functionality **in** and **outside** the scope. The classic context diagram from Structured Analysis (SA) [DeMa1979] is often used as a means of representation but today—because there are hardly any tools to support SA—many other diagram types with equivalent content can be used (e.g., a UML class diagram, a use case diagram, or a component diagram).In addition, a tabular representation can be used as a substitute for a context diagram as long as the basic elements listed below are present.

2.2.1 Basic elements of context diagrams

The three essential basic elements of a context diagram are:

- The system under development (more precisely, the system boundary)
- Neighboring systems or actors of the system under development (all people, roles, IT systems, equipment, etc. with which the system has interfaces)
- The (logical) interfaces between the system and its neighboring systems

Experience shows that the interfaces between the system and the context can best be determined by the incoming and outgoing data. The classical context diagram therefore focuses on this input and output data from and to neighboring systems. In this sense, the context diagram is the most abstract form of a data flow diagram (see Section 4.3) because the complete functionality of the system is reduced to one function (namely the whole system). The focus of this diagram is the identification of all interfaces of the system under development.

2.2.2 Example of a context diagram

Figure 6 shows an example of a context diagram using Structured Analysis. The overall system (an early warning system in the mining industry) is represented as a circle in the middle. The human neighboring systems are shown in the example as stick figures and the organizational and technical neighboring systems as boxes. The interface is modeled in the form of data flows to and from the neighboring systems.

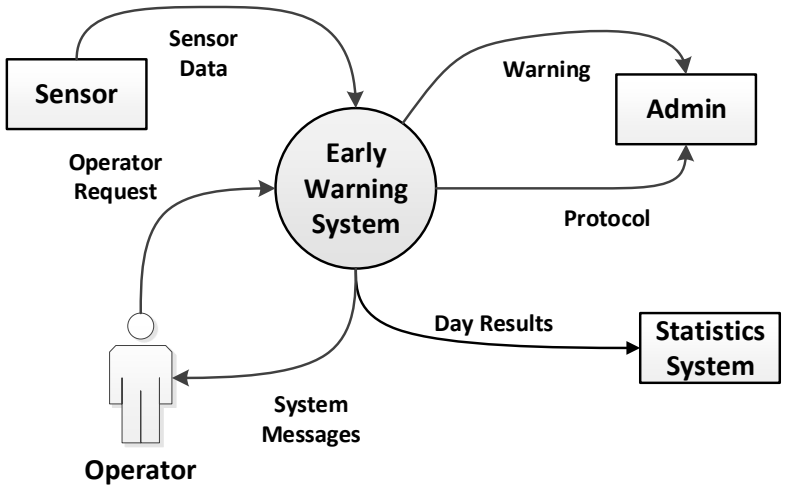


Figure 6: Example of a context diagram

Today, SysML block diagrams [OMG2010a] can be used to model the system context, for example. Figure 7 shows the context diagram of an automated machine for the production of cylinder heads for cars (see [DaTW2012]).

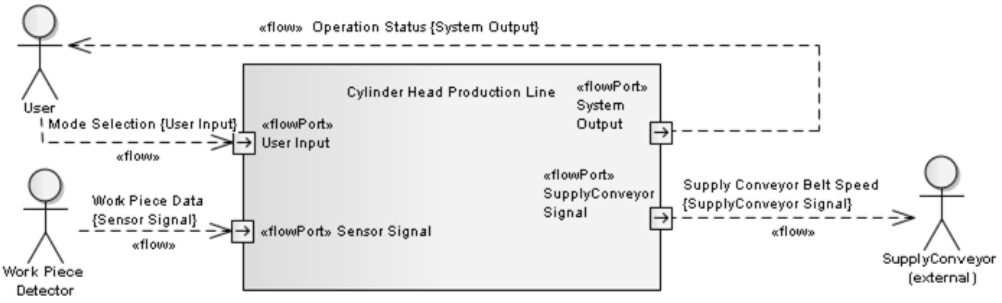


Figure 7: Example of a context diagram in SysML block diagram form

The diagram shows actors in the system context and the data flows between actors and the system under development. Such context diagrams based on SysML document very similar information about the system context to context diagrams which are based on the data flow diagrams of Structured Analysis.

2.2.3 Notation elements for modeling context diagrams with data flow diagrams

Data flow diagrams can be used to model data flow-oriented context diagrams. Figure 8 shows possible model elements for the construction of data flow-oriented context diagrams based on data flow diagrams according to DeMarco (cf. [DeMa1979]).

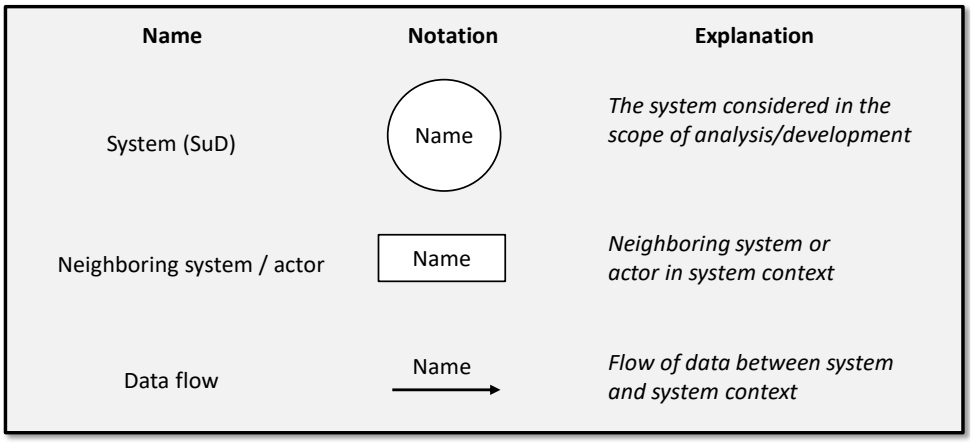


Figure 8: Possible modeling constructs of data flow-oriented context diagrams

In context modeling using data flow diagrams, the system under development is often represented by a circle, sometimes a box or a cloud. The corresponding modeling construct represents the system under development, which, for example, represents either a part of a company, a business process, or a system to be automated. It thus expresses the scope of the system under development (i.e., the system boundary). The presentation of the neighboring systems is relatively arbitrary; often these are modeled as boxes but can also be modeled as stick figures or as a 3D box or as double lines for external databases or "files".

In Structured Analysis according to DeMarco, neighboring systems (sources and sinks) are called terminators (= terminals). Neighboring systems or actors represent any kind of communication end points of the system under development. Neighboring systems or actors can on one hand be people who work with the system, but on the other hand hardware/software systems, devices, sensors, actuators, or passive data storage (such as databases or files)—that is, everything or everyone who delivers input to the system or receives output from the system (or both). The neighboring systems thus represent parts of the context of the system under development.

The data flows between neighboring systems or actors and the system under development represent input and output interfaces of the system under development. These data flows are mostly shown as straight or curved lines with an arrowhead to the system (for input), arrowhead to the neighboring system (for output), or as a double arrow. Data flows in this type of context diagram represent the incoming and outgoing data or control information. Mostly, these arrows are interpreted as data flows into or out of the system. If control flows are represented in this way, this should be explained in a legend to the diagram.

2.2.4 Pragmatic rules for context modeling with data flow diagrams

The following pragmatic rules should be considered:

- All neighboring systems that interact with the system should be included in the diagram (completeness of the communication partners).
- All neighboring systems should be named (to clearly specify where the input comes from and where the output goes to).
- All inputs and outputs should be labeled with the logical name of the data flows (because unnamed arrows indicate a lack of understanding of the interface).

2.3 Other types of context modeling

The cooperation between the system under development and the neighboring systems in the context is also the subject of the use case view (see Section 4.2) and the scenario view (see Chapter 5). In addition to defining the system boundaries (scoping), the use cases are used to roughly structure the system's functionality. With the scenario view, sequences of communication and other communication details can be specified more precisely in addition to the specification of the data flows. Current research includes proposals for context modeling in a state-oriented view, in which the state of the system context and corresponding state transitions are modeled.

There are also approaches for modeling static-structural aspects of the system context by using information structure view diagrams. Other approaches to context modeling consider the system in the context of a data flow-oriented view by modeling functions in the system context (context functions) and documenting their relationship to functions of the system.

Such approaches are used in particular for mechanical detection of unwanted functional interactions between the system and its context (*feature interactions*). An overview of the different types of context modeling in requirements engineering can be found in [DaTW2012].

2.4 Further reading

Data flow-oriented context diagrams

- DeMarco, Tom: Structured Analysis and System Specification, Yourdon Press, Prentice Hall, 1979.
- Daun, M.: Tenbergen, B.; Weyer, T.: Requirements Viewpoint. In: Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M.: Model-Based Engineering of Embedded Systems, Springer, Heidelberg 2012.

Use case-oriented context diagrams

- Jacobson, I.: Christerson, M.; Jonsson, P.; Oevergaard, G.: Object Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Reading, 1992.

3 Information structure modeling

3.1 Purpose

The modeling of information structures has a central role in requirements modeling, mainly because it has two tasks:

- Specification of technical terms and data
- Specification of requirements that relate to technical terms

A glossary is often used to define technical terms in requirements engineering. In a glossary, the meaning of the terms in the domain or in the language of the client is defined. With the introduction of information models, the content of a glossary is supplemented with important information. Information modeling often starts by looking at all nouns that occur either in textual requirements, or, for example, in data flow-oriented or control flow-oriented requirements modeling in the naming of functions of the system (see Section 4.3).

In an information model, however, a lot of emphasis is placed on the relationships between the terms. Expressing these relationships is one of the strengths of diagrams of the information structure view compared to a textual, perhaps alphabetically arranged glossary. The second step is to define the "attributes" of the terms. Attributes express the relevant properties and technical information of a term. Thus, relevant properties can be clearly represented in an information structure diagram—for example, for a customer in a CRM system. With this kind of information modeling, a conventional glossary is expanded to include additional information. The glossary can be derived automatically from this type of diagram. Thus, the use of information models also fulfils the purpose of a glossary—the definition of terms that should be used uniformly throughout the system development.

Another use for the modeling of information structures is the precise specification of requirements. All information modeled in the structures should be considered as requirements (see also Section 1.3). The statement above, about which customer data is relevant for a CRM system, can also be interpreted as "data that the CRM system must manage for a customer".

3.2 Modeling information structures

This section looks at the requirements in the information structure view using UML class diagrams. There are several approaches for modeling information structures. One diagram that is related to this kind of modeling is the ER (entity-relationship) diagram [Chen1976]. Today, it is commonly used for modeling database schemas. The relationship with the class diagram consists in the transition from a (logical) information model in requirements engineering to a physical database schema. The information model is a good basis for designing database schemas, that is, the storage of business data.

The great advantage in the use of UML class diagrams lies in the UML integration with other diagram types that are used in other views in requirements modeling (see Section 1.5). This can be necessary to achieve the links required for a formally correct, complete, and understandable requirements model—for example, the link between activity diagrams and the information model.

This integration also determines the approach for the creation of an information model within the framework of requirements engineering. Usually, you will create such a model to have a good basis for modeling other views. However, it quickly becomes clear where the deficits lie in the information model. In this case, any deficiencies in diagrams or other views because, for example, when the functions were defined, not all required technical information was considered, are then identified. This change between the different perspectives is not always easy but has great potential with respect to the correctness and completeness of the modeled requirements.

3.3 Simple example

The figure below shows a simple example of a data diagram in the form of a UML class diagram. It shows the relevant terms, the attributes, and the dependencies.

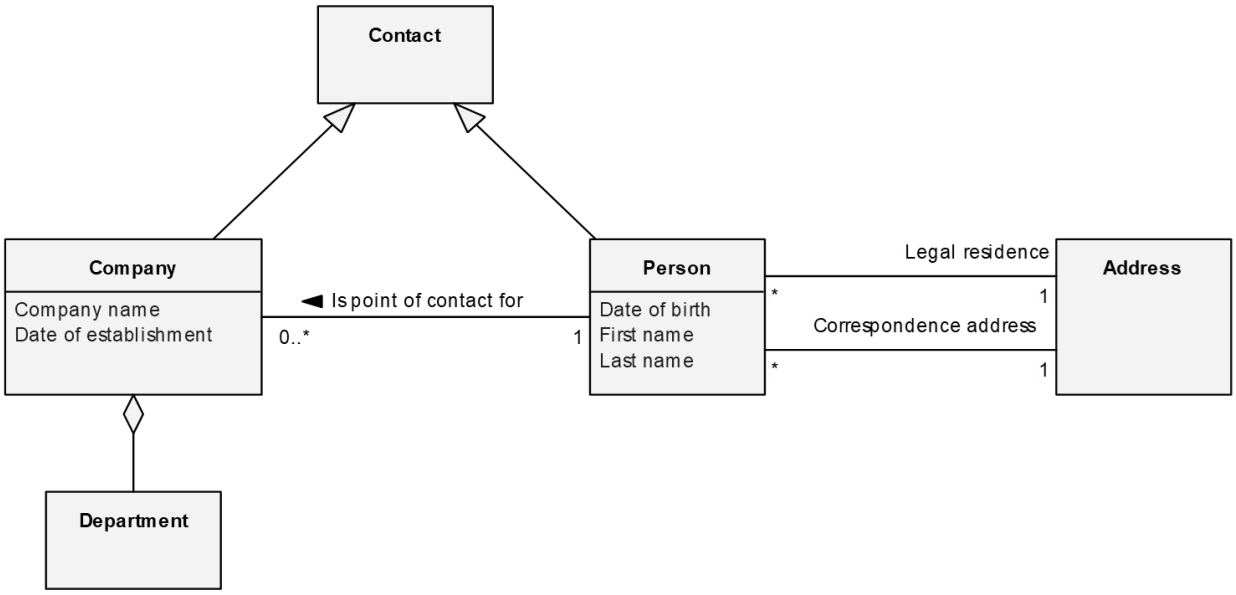


Figure 9: Example of a class diagramm

The above class diagram consists of five classes: *contact, company*, *person*, *address,* and *department*. It documents the essential properties of these classes in the form of attributes—for example, the attribute "*date of birth*" of a person—and the dependencies between these classes, such as that a person is a representative for a company or that a company is made up of departments.

The meaning and use of the various modeling methods of class diagrams are considered in detail in the following sections.

3.4 Modeling classes, attributes, and data types

The central element of information structure diagrams modeled on the basis of UML class diagrams are the class and the attributes of the class.

3.4.1 Classes

3.4.1.1 Objects versus classes

When information structure models are used in requirements modeling, two terms must be differentiated: objects and classes. A "class" is a pattern or template which defines the common properties of many objects. The objects are then referred to as instances of these classes.

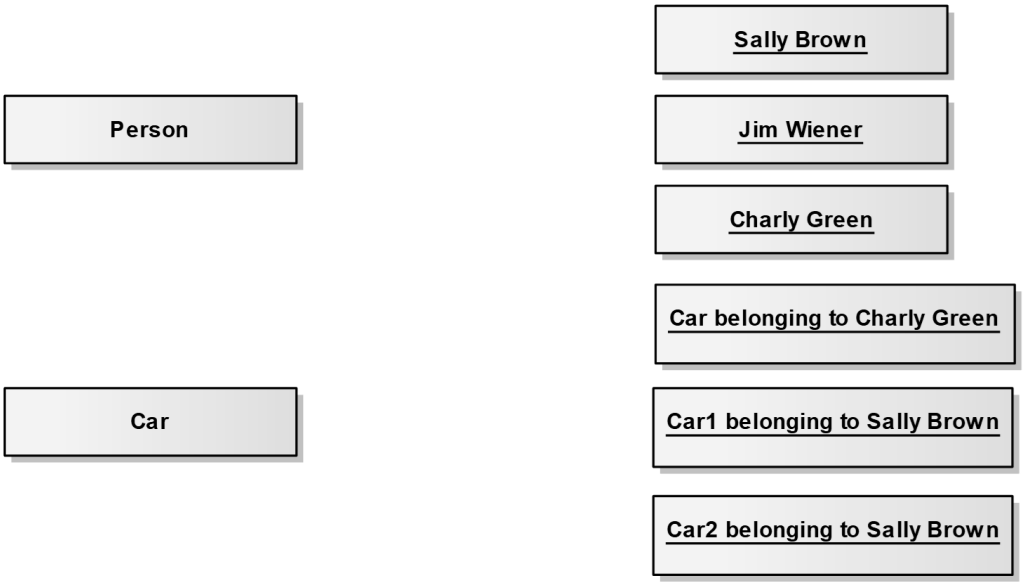


Figure 10: Class vs. object

Figure 10 shows the classes *person* and *car* and on the right, some objects as instances of these classes. For these objects, an important property of the objects is also shown: they are unique and should therefore also have a unique *identifier* (for more information about uniqueness, see Section 3.4.2). With the unique name in the figure above, the two cars belonging to Sally Brown can be differentiated.

3.4.1.2 Syntax and semantics

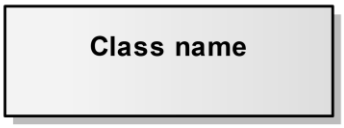


Figure 11: A class

The simple representation of a class consists of a rectangle with the class name. This is expanded in Section 3.4.2 with the representation of attributes.

As mentioned above, a class represents the template for a plurality of objects of this class which are referenced in the requirements. Therefore, in general, the name of a class is used in the singular. When referring to a person, the class name "*persons*" would be incorrect as this means multiple persons.

The statement that a class represents the template for a plurality of objects of this class is a general statement for a class diagram. You can, however, formulate the data structure perspective of a requirements model more easily with the class diagram: the terms that are relevant in the domain in question appear as classes in the diagrams of this view. In other words, the nouns that are used in the formulation of the requirements appear as classes.

With the distinction made above between an object and a class, the latter needs to be clarified because the requirements (textual or graphical) are terms used to refer to any object of that class.

Example: **The system must display the data of a person**
Assume that in an information model a class person exists. This requirement is to be interpreted such that the data for each object of the class person is to be displayed.

This results in the first task of modeling the information model: identifying the required classes from the objects used in the requirements.

3.4.1.3 Heuristics for identifying classes

One of the simplest approaches for identifying classes is to define a class for every noun in the requirements (or the current specifications). However, you will quickly find that this approach provides a vast number of classes which then have to be processed further. Many of the classes found only describe the properties of another class. These classes are then added to this other class as class attributes (see Section 3.4.2). Another aspect of reducing the vast number of classes is to classify synonyms or phrases out of context, for example.

Let us assume that the following nouns would have been identified in a first step: person, age, car, gender, color, vehicle, man. In this list, there are only two terms that are worth

modeling as classes (cf. [Mart1989], [ShMe1988]): *person* and *vehicle*. For the other terms, the following applies:

- *Man*: synonym for person
- *Age*: property of a person
- *Car*: synonym for vehicle
- *Gender*: property of a person
- *Color*: property of a vehicle

With this selection, three assumptions were made that need to be confirmed in the context of a real development project:

- The concept of *person* must be used consistently and not man.
- The concept *vehicle* must be used consistently and not car.
- The term *color* refers to the color of a vehicle.

For synonyms, the common language use of the project or a company is decisive—as long as it is unique. This procedure allows a good first version of the information model. Further heuristics that extend the approach presented are described in Sections 3.4.2.2 and 3.6.3.

Another way to find classes is to search directly for specific candidates in typical formulations. These can be divided into three areas:

- Tangible or intangible objects
- Roles
- Functions

This procedure significantly reduces the set of all nouns.

3.4.1.4 Tangible and intangible objects

Tangible objects in the real world are relevant for the requirements as they are either affected by the system under development or have a "representative" (e.g., a class) in the system under development (or both cases can apply).

Examples:
person, car, door, book, leave application (which is not printed, so does not have to be tangible) or *club*.

3.4.1.5 Processes

To support the system processes, additional and relevant information is often needed, such as: *delivery*, *order*, *call*, *assembly,* or *report*. For example, the data of a delivery, such as the date of receipt or the agent, may be technically relevant to the system.

Note that the term in the information model is not the function to be implemented by the system. The information model describes the relevant information for the process—not the process itself which is to be supported by the system (see also Chapter 4). This process is generally denoted by a noun in combination with a verb in its normal form, rather than only by a noun, as is the case in the information model.

Depending on the field of application, an order could be a useful class in the information model. The receipt of an order could then be a supportive function of the system. It can be used to derive, for example, the names of use cases (see Section 4.2): *receive order, forward order,* and *complete order*.

3.4.1.6 Roles

Similar to functions, roles of objects can be interesting for information structure models. These roles are then defined as separate classes.

Examples are:
- Driver: a person in the role of the driver of a car
- Residence: the address of the first residence of a person

There is another alternative for modeling roles in the information model. More information about this alternative can be found in Section 3.5.1 and Section 3.7.1.

3.4.1.7 Defining the meaning of terms

An important property of an information model is that the terms defined in the model are placed in context (see Section 3.1). Together with the definition of the attributes, this means that a large part of the meaning is generally already defined. If additional descriptions are necessary, textual additions can be defined, which are then placed in a relationship with the corresponding class.

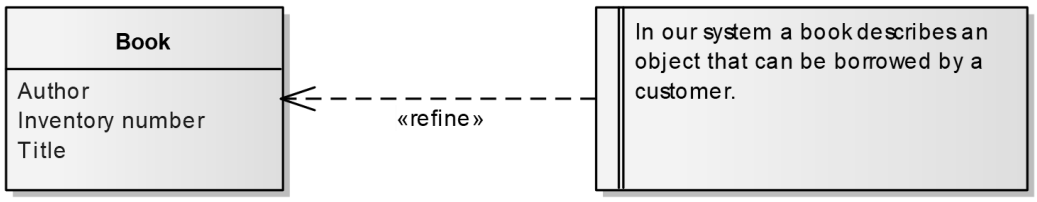


Figure 12: Class and natural language definition

3.4.2 Attributes

Attributes are used to specify classes more precisely, which means that defining attributes enriches the corresponding diagrams with additional semantics. This is very important in requirements modeling.

3.4.2.1 Syntax and semantics

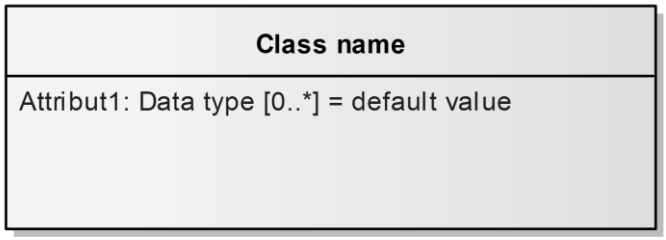


Figure 13: Class with attribute

The attributes are defined within the scope of the class. The following components are allowed (represented in Backus–Naur form):

[/] Name [: type] [multiplicity]] [= default]

- **Name**: the name of the attribute, which is obligatory
- **Data type**: the data type of the attribute; this is optional and is described in Section 3.4.2.4
- **Default**: the value of the attribute set on creation of a new object of the class
- **Multiplicity**: can be used if the attribute can take on multiple values simultaneously (e.g.: several first names); the same multiplicities are used as in the relationships (see Section 3.5)
- **Derived**: the leading "/" indicates that the attribute value can be derived from other values (e.g.: the age of a person can be derived from the date of birth)

The attributes specify domain-specific properties of a class that are relevant for the system under development.

3.4.2.2 Heuristics for determining attributes

To distinguish between classes and attributes, check each noun which was found as a potential class (see Section 3.4.1). In each case, consider whether the noun is merely a property of another class. If so, this noun is defined as an attribute of this other class.

Attributes are often identified as such because of wording in written or spoken text. Common types of formulations that indicate potential attributes of classes are the following:

Noun in combination with a genitive

Example:
- the date of the order
- the diameter of the circle
- the color of the car

The names of the attributes and the corresponding class are already given in the formulations. No further interpretation of the formulation is required.

`Sentence construction with: <class> has <attribute>`

Example:
- a person has a date of birth
- an address has a postal code
- the process has a transition time of ...

This type of formulation is an indication of an attribute of a class or a relationship between two classes. More information about the distinction between whether something is an attribute of a class or a relationship between classes can be found in Section 3.4.2.3.

`Adjective in combination with a noun`

Example:
- a fast car
- a large display
- a huge bank account
- a red car
- a black list

This type of formulation usually indicates a concrete instance of a class (car ☒ fast). We have to determine which attribute of the class is meant (e.g., size of display = large) (see Figure 14).

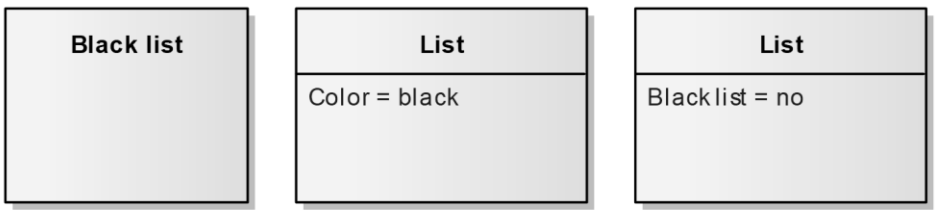


Figure 14: Modeling variations for adjectives with nouns

Sentence structures with: <class> is <attribute value>

Example:
If the person is an adult; if the application is approved; ...

In this case, only a value of an attribute is specified. Again, further analysis is necessary because in the examples above, classes are compared with attribute values. However, the values apply to attributes of the class and not to the class itself (e.g., approved is a value of application status).

Differentiating objects

In addition to the formulations presented, attributes can also be derived from a required property of objects in the object-oriented paradigm: objects always have to be unique in their context.

This uniqueness must be achieved by using different values of the attributes of objects. At any time, the combination of the attribute values must be different between objects of the same class. Only then can the objects be uniquely distinguished for a user of the system.

Example:
Modeling the object Peter Schulz with only two attributes (first name, last name) may not be sufficient to distinguish it from another person with the same name. If the class person also has the date of birth as an attribute, its objects may be clearly distinguishable (i.e., another person with the same name but born on a different day).

3.4.2.3 Class or attribute

The distinction between a class and an attribute is not always easy. If there is any doubt as to whether an identified term should be represented in the information model as a class or an attribute, then the term should first be modeled as a class. In contrast, if the term identified is simple, unstructured data such as text, dates, numbers, or Boolean information, then the term should be represented as an attribute in the information model.

For structured information, the following heuristic is helpful: as soon as a structured form of this information belongs to more than one other object, it should be modeled as a separate class.

The example in Figure 15 shows the difference for an address. Objects of the class *address* can belong to multiple objects of the class *person*. These objects share an address. Changes to an address affect all *persons* that are associated with that address. In contrast, the addresses in the second part of the example are completely independent.

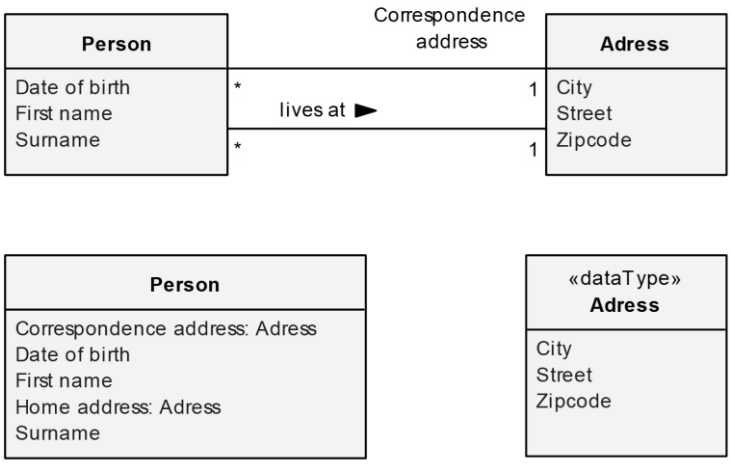


Figure 15: Class or attribute

3.4.2.4 Information modeling for existing systems

Existing systems have a rich pool of resources that can be used to create an information model. They help to identify not only classes and attributes but also relationships and multiplicities.

Possible sources:

- Logical or technical information model (entity-relationship models)
- Interface specification
- Description of a data warehouse

On one hand, the challenge with this existing information is—as with any system archeology—that the information has to be validated and checked for accuracy. On the other hand, we should avoid including technical implementation attributes (technical identifiers and optimizations) in an information model.

3.4.3 Data types

Requirements modeling with UML class diagrams distinguishes between three kinds of data types: *primitive data types*, *structured data types,* and *enumerations*.

3.4.3.1 Syntax and semantics

The syntax for data types is similar to the syntax for classes. The name is mandatory. Further information can be added to determine the allowable set of values of attributes.

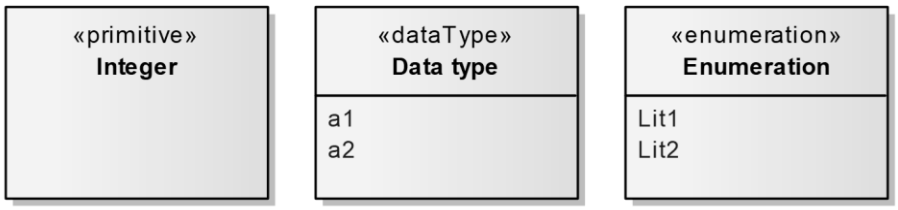


Figure 16: Examples of data types

Primitive Types: unstructured data types

The primitive data types are unstructured and thus the simplest data types. They represent simple data types such as a number, Boolean value, string, etc.

UML has a number of pre-defined primitive data types:

- **Boolean**: a Boolean value, can be TRUE or FALSE
- **Integer**: a whole number
- **Float**: a floating point number
- **Character**: a single character
- **String**: a sequence of characters

Depending on the application, it may be useful to specify more primitive data types, that is, to define data types that do not require more in-depth definition.

Example:
String50. It is clear, without further description, that a string of length 50 is meant.

Structured data types

This kind of data type allows the definition of structures, that is, the definition of complex data types that are composed of more simple data types. These are always very specific to a certain application area. UML specifies only the mechanism for defining such data types and therefore does not contain any concrete data types. Figure 17 shows several examples.

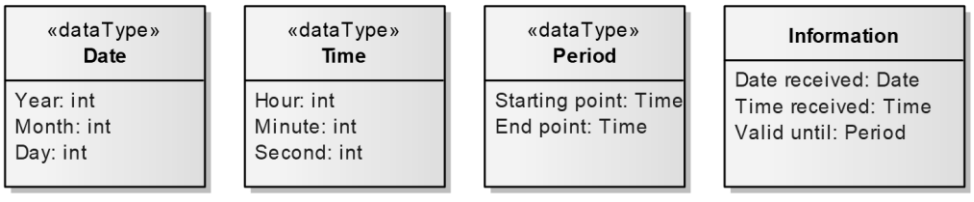


Figure 17: Example for the modeling and use of data types

As the example in Figure 17 shows, these data types can be defined hierarchically. The end point of the hierarchical definition is primitive data types or enumerations.

Enumerations

If the domain of an attribute can be specified by a denumerable list of acceptable values, this data type can be defined as an enumeration. Figure 18 shows two examples of the definition of an enumeration type.

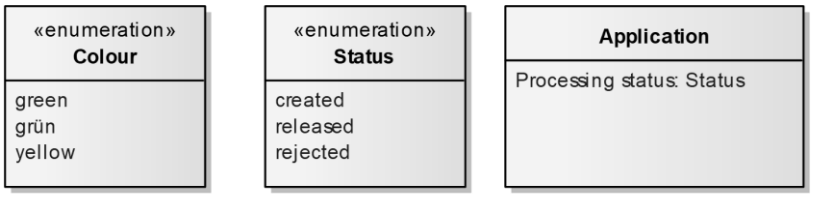


Figure 18: Enumerations

The above example is a typical case of the use of an enumeration: the definition of a status (for an application). However, the definition of this data type is redundant when a state machine for the class "application" is available (see also Section 0). Therefore, only one of the two should be included in a requirements model.

3.4.3.2 Heuristics for determining data types

When creating an information model during requirements engineering, we have to decide whether it is useful to model the data types of attributes of a class at this point in the project. The advice here is to model a data type immediately (preferably a primitive data type).

During further modeling, this can be redefined or refined into a more complex data type, or even a stand-alone class as required. If necessary, the data type can be specified in more detail by textual requirements.

The next question would then be to identify more information about the data type. For enumerations, the answer is obvious: we identify the possible values of the attribute and list them in the enumeration. For structured data types, the necessary information is found in the domain of the application. This is similar to the question for identifying the necessary attributes of a class (see Section 3.4.2).

3.4.4 Recommendations for modeling practice

3.4.4.1 Modeling tip: attribute constraints and textual requirements

If the UML options are insufficient or the results are not "easy to understand", we can add textual requirements.

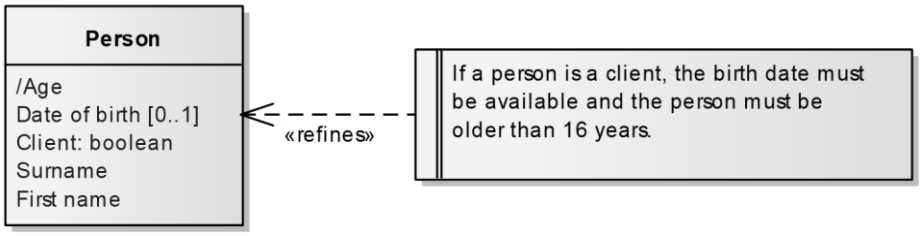


Figure 19: Modeling attribute constraints

3.4.4.2 Modeling tip: views of things

In the language of project stakeholders, a term is often used implicitly for several things or views of one thing (homonym). For example, *the request* may be used as a homonym for: *the empty paper form*, *the completed document,* and *the signed document* or *the data in the system*. The diagram must clearly state which meaning the modeled terms have. Stereotypes may help to clarify the situation.

3.4.4.3 Modeling tip: length vs. number of strings

When attributes of a class which contain text are defined (e.g., a person's name), then the question of the maximum length of the string arises. Multiplicity is often misused in this case. According to UML, *first name:string[20]* means there are 20 first names of the type string. This does not define a string of length 20. We can resolve this ambiguity problem in UML by defining a special data type.

3.4.4.4 Outlook: specification with OCL

For the exact definition of constraints, OCL (Object Constraint Language) from OMG [OMG2012] provides the possibility of a more formal specification which, however, is not always easy to understand. The condition that a customer must be 16 years of age or older could be formulated as an OCL constraint as follows:

```
context Person inv: self.Client=true implies self.age >= 16
```

3.5 Modeling relationships

A key component of an information model is the relationships. They are represented as a connection between classes and express how (i.e., with what meaning) the objects of the specific classes are related to each other. The most commonly used relationships in the modeling of requirements are simple relationships (binary associations), aggregations, and compositions.

3.5.1 Simple relationships (binary associations)

Simple relationships are drawn between classes and describe the relationship which **two** objects have to each other. The two objects can thereby be instances of two different classes or of the same class.

In addition to simple relationships, UML provides n-ary relationships which connect multiple objects. However, these are not discussed further in this document.

3.5.1.1 Syntax and semantics

Binary associations are modeled as a line between the corresponding classes. In order to give this line a meaning, additional information is added. Figure 20 considers the classes *person* and *address*. The model should state that a person has exactly one address assigned where they live and also exactly one other address to which correspondence should be sent. An address can be assigned to more than one person as the correspondence address or residence.

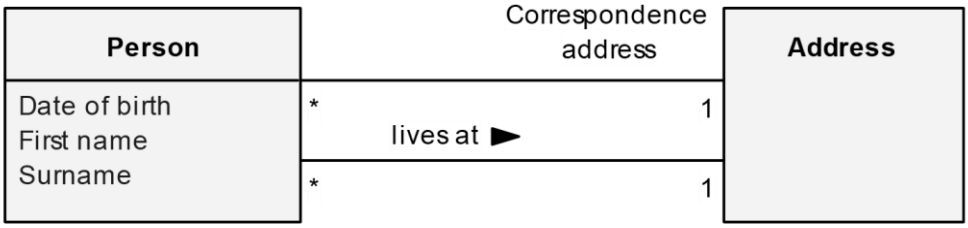


Figure 20: Example of modeling simple relationships

- **Name**: Specifies the name (meaning/semantics) of the association in a verb phrase
- **Reading direction**: Direction in which the name is to be read
- **Multiplicity**: Is listed at each end of the association and indicates how many objects the other object may or must be related to
- **Role**: Refers to the role played by the object to which the role is attached with respect to the other object

To identify this additional information for relationships, it is helpful to imagine the objects, especially when determining multiplicities.

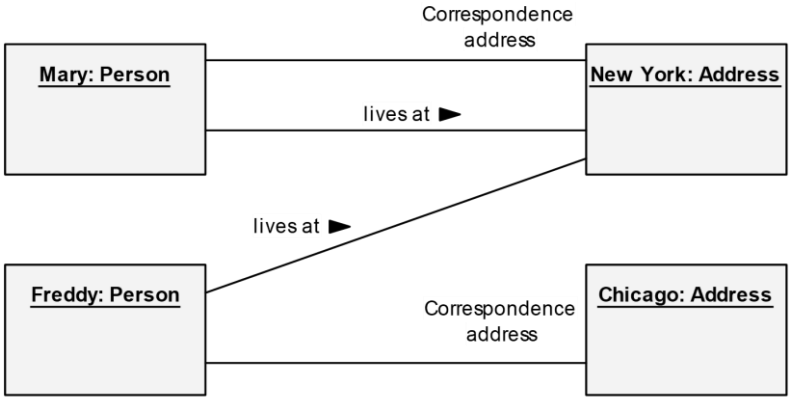


Figure 21: Relationships of the objects

In addition to the requirements contained in the information model, associations are often the basis for deriving functional requirements.

Example: Requirement without the use of associations
Show address

A functionality, as in the example "Show address" above, which refers to only one object ("Address") without considering its relationship to other objects, is often incomplete. Relationships are very useful for defining the context precisely and thus reducing the set of objects to the desired/required quantity.

Example: Requirement with the use of associations
Show the correspondence address of the person who is the contact for the company

Associations offer the opportunity to move through the information model. This ability to navigate through the information model also shows the importance of the unique name for the associations between classes, especially when multiple relationships exist between two classes. For this purpose, we refer to either the name of the association or a role at the end of the relationship. When formulating requirements, role names can be used instead of the class names (see the example and Figure 9).

For a requirements engineer, multiplicities are an important tool for verifying the details of the quantifiers in the requirements:

Examples:
- Requirement 1: Show the person
- Requirement 2: For this person, show the company for which it is the contact person

The formulation of requirement 2 seems to assume that there is exactly one legal entity. The multiplicities in the diagram show a different picture.

For a requirements engineer, the following questions regarding the requirements and the association arise: Is the multiplicity of the association correct? If it is incorrect, it must be changed. If it is correct, then the following questions must be answered:

- What should happen if a legal entity is assigned?
- What should happen if more than one legal entity is assigned? How is the one you want to display selected (e.g., the one with the youngest or oldest date of incorporation)?

3.5.1.2 Heuristic for determining simple relationships

Linguistic Formulations

Relationships between classes can be discovered by certain statements in the natural language. Statements such as "A departmental manager manages a department" can be expressed directly in the diagram. Depending on the formulation of such statements, they are drawn in different ways in a class diagram:

Verbs → binary association, association name, read direction

"Head of department **manages** *department" or "Departments* **are managed** *by departmental heads"*.

Verbs in an active or passive formulation indicate the meaning of the association. In a model, verbs in active form are preferred. When requirements are the basis for the determination, then verbs (= functionality) must be critically queried.

Example:
Employee orders product

In the information model, this would only be included as an association if the information about which employee has ordered which product is relevant.

Nouns → role

"Employee is **head of** *a department"*

If two concepts are connected with a noun, then it is usually a role that sets one of the two terms over the other. If the role contains properties, then this role could also be modeled as a separate class (see Section 3.7.1)

Quantifiers → multiplicity

"A natural person **can** *be a contact for* **any number of** *legal entities"*.

*"For a legal entity***, exactly one** *natural person is the contact"*.

Quantifiers specify the associations found and are absolutely necessary for both ends of the relationship. A statement mentioning "a/one" should always be questioned with "exactly one?".

Classes without further reference in the class diagram

Each class in the information model must be in a relationship with at least one other class (via a simple relationship, generalization, an aggregation, or a composition). If classes exist that are not in a relationship with any other class, this gap needs to be closed. This means that the classes and the relationships between them form a network.

3.5.2 Aggregation and composition

For certain types of relationships (more precisely, the semantics of relationships), UML has specific notation elements.

3.5.2.1 Syntax and semantics

In UML, a "part/whole" relationship can be represented with a line on which a diamond shape is located at the end with the class that represents the whole.

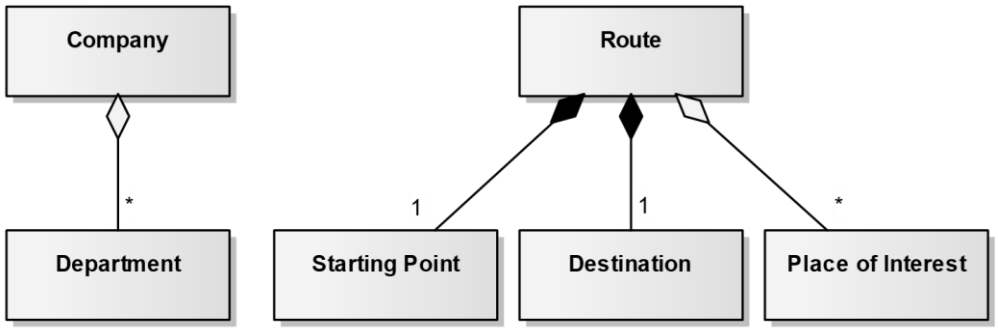


Figure 22: Example for the modeling of aggregations and compositions

This is primarily a relief when modeling and reading the diagrams because the importance of the association is clear immediately. A special form of aggregation is the composition. Here, the part/whole connection is particularly strong. It is used to specify that deleting the whole also deletes the parts.

3.5.2.2 Heuristics for determining aggregations

Because aggregations and compositions are considered as specific types of a relationship, the heuristics for identifying relationships (see Section 3.5.1) can also be used to identify aggregations and compositions. From the perspective of the specific meaning of such associations, aggregations and compositions are indicated by keywords that relate to statements about part/whole dependencies.

Verbs

Typical verbs that indicate aggregation or composition relationships are:

- consists of
- is composed by
- contains
- results
- has

Example:
"A company consists of departments "

Nouns

Aggregations and compositions can also be identified via role formulations. Depending on the meaning of the relationship, these are:

- part
- whole
- component

Example:
"A department is part of a company"

3.5.3 Association classes

3.5.3.1 Syntax und semantics

A mixture of association and class is the so called association class. By using association classes, it is possible to allocate properities directly to concrete associations between classes.

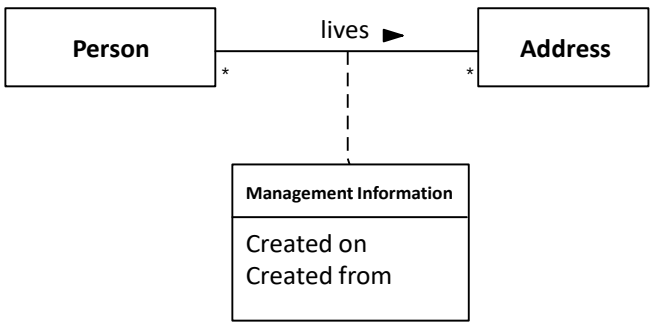


Figure 23: Simple Example of modeling management information with association classes

In the example shown above the link between an object of the typ "Person" and a particular object of the typ "Address" has been extended by an object of the type "Management Information". The object of the type "Management Information" enriches the association by adding the information when and who has created the corresponding relationship. In this case, to any relationship between objects of the type "Person" and "Address" an additional object exists holding the corresponding management information. Due to the semantics of association classes no additional multiplicities are modeled.

The modeling of assocation classes is controversly discussed as novice user interpret such models often in a wrong way. In doubt and in order to validate the interpretation such diagrams can also be modeled with normal classes and associations between them.

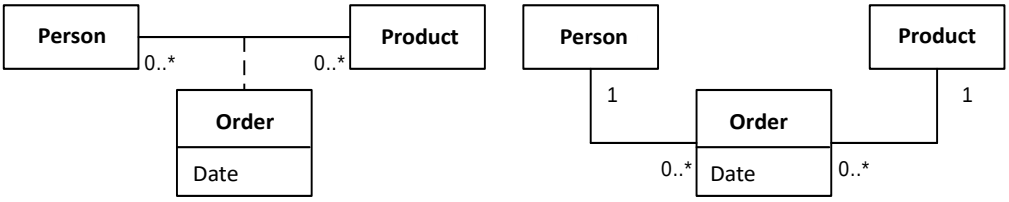


Figure 24: Transformation of modeling of association classes by using "normal" classes

The example at the right hand side in the figure above is sometimes misinterpreted as: A person can order several products when placing an order. For a better understanding Figure 25 shows a valid example for the instantiation of the class diagram displayed at the left hand side of Figure 24.

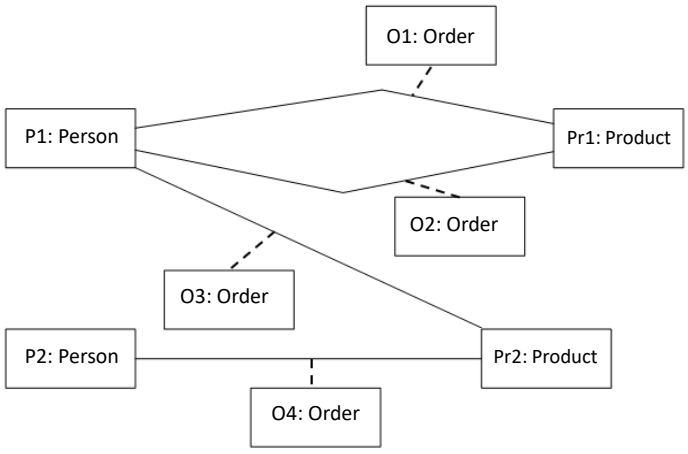


Figure 25: Exaple for a valid instantiation of the class diagram in Figure 24 (left hand side)

The example shown above can be extended concerning the fact that a person can order more than one items of a particular product. For instance, by adding an attribute Quantity to the class Order.

3.5.3.2 Heuristics for identifying association classes

Association classes attribute associations. Linguistically, all formulations that refer to properties concerning an association are interesting.

> **Example:**
> How long / since when a person lives at an address.
> When / how often a person has visited a place.

3.5.4 Practical advice for information modeling

3.5.4.1 Modeling tip: constraints of relationships and textual requirements

If the UML options are insufficient or the results are not "easy to understand", then we should use textual requirements in addition to the model.

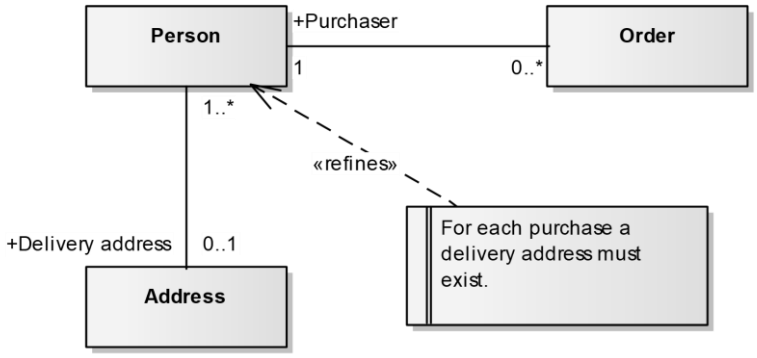


Figure 26: Modeling constraints of relationships

3.5.4.2 Modeling tip: attribute or association

Two classes that are connected to each other with a 1:1 or 0..1 relationship can occur but this situation is rather unusual. In this case, we should question whether one of the two classes can be converted into an attribute of the other class.

3.5.4.3 Modeling tip: navigability vs. reading direction

When modeling classes, there are two representations of relationships that can be interpreted as "directions" with a very different meaning (not counting the triangle of the generalization that could also be misread as a direction arrow). One representation is the reading direction of the name of the association (i.e., the small arrowhead next to a verb) (see Section 3.5.1), as shown in the upper part of the following figure.

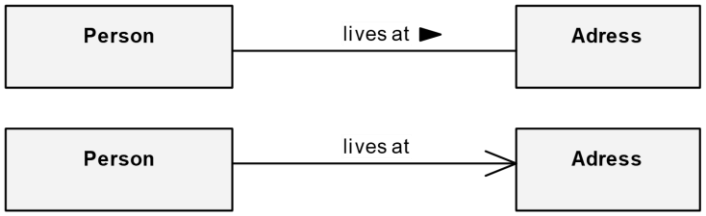


Figure 27: Reading direction vs. navigability

The other representation is the navigability as shown at the bottom of the figure above. The latter states that for a person, we can get the address at which he resides but not vice versa. This navigability is important in the realization. In requirements engineering, however, it plays a minor role.

3.5.4.4 Modeling tip: different interpretation of multiplicities (versioning, historizing, dynamics)

Multiplicities appear to be defined very precisely. However, they can lead to discussions or different interpretations.

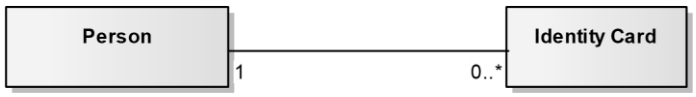


Figure 28: Unclear multiplicities

0..* can be interpreted as:

- *: Person has (over time) many identity cards (expired, lost)
- 0: does not need an identity card (does not have one or has lost it)
- 0: A person always has an identity card but the person is created first and then the card. Therefore, there is a period before the identity card is created when a person exists without an identity card.

An information model always shows a static and consistent structure of the information. Accordingly, there is no intention to resolve intermediate states of the information. Other temporal aspects, such as versioning or history, may well be relevant and modeled accordingly. Figure 29 shows a possible modeling of a simple history.

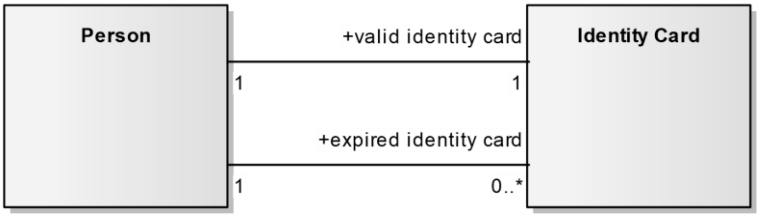


Figure 29: Resolution of unknown multiplicities

3.5.4.5 Outlook: specification with OCL

For the exact definition of constraints, OCL (Object Constraint Language) from OMG [OMG2012] provides the possibility of a more formal specification which, however, is not easily understandable. The condition that each person in the role of purchaser must have a delivery address could, for example, be expressed by the following OCL constraint:

```
context order

inv:self.purchaser->notEmpty()implies self.Purchaser.DeliveryAddress-
>notEmpty()
```

3.6 Modeling generalizations and specializations

3.6.1 Syntax and semantics

The common properties and relationships of multiple classes can be summarized by a generalization. Models can thus be simplified. The corresponding classes are connected with a line with an arrowhead at one end. The class that the arrowhead points to represents the generalized concept. If the class has no objects (i.e., no instances of this class), then it is called an abstract class. To illustrate this in the diagram, the name of an abstract class is displayed in italics. Figure 30 shows a simple example for the modeling of a generalization.

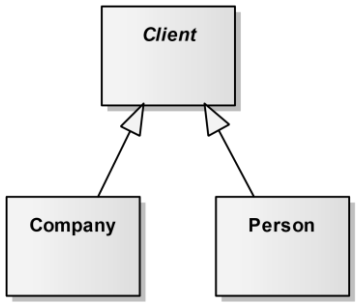


Figure 30: Example for the modeling of a generalization

Generalized terms should be used with caution, as there is a risk of misunderstandings. Abstract and non-abstract generalizations have a different meaning for requirements.

In this context, abstract generalizations are—in contrast to non-abstract generalizations—representative of each of their specializations.

The system must provide the user with the ability to create *clients* <abstract generalization>

This corresponds to:

1. The system must provide the user with the ability to create companies <Specialization1>
2. The system must provide the user with the ability to create persons <Specialization 2>

When "Client" is not an abstract class (i.e., it is not italicized), the above requirements allow the creation of a client object (without specifying whether the client is a company or a person).

3.6.2 Generalization sets and their constraints

Generalization sets offer the option of combining different aspects of a generalization to form groups of subtypes. Figure 31 models two generalization sets (*contact kind* and *contact type)* with associated constraints.

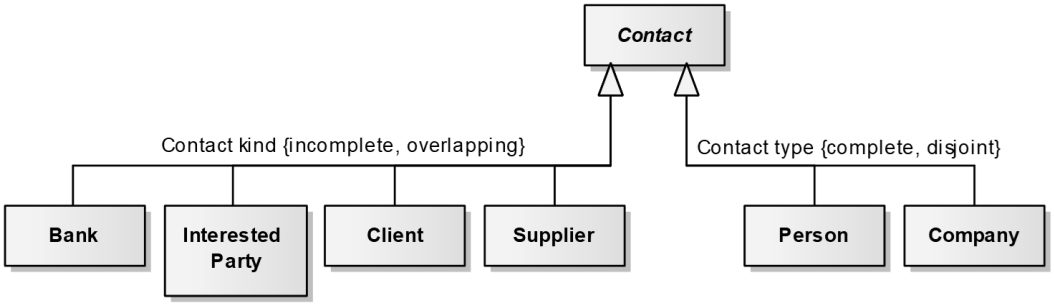


Figure 31: Example for modeling generalization sets and constraints

In UML, the specification of properties of such a generalization set is annotated by constraints in curly braces.

Typical constraints are:

- **Incomplete**: The modeled subtypes are not necessarily complete. For example, manufacturer could be added as a contact kind.
- **Complete**: The modeled subtypes are complete. No other contact types are possible.
- **Disjoint**: An instance can only be one of the subtypes. For example, a contact is either a person or a company, but never both.
- **Overlapping**: An instance can belong to more than one subtype. For example, a contact may be a customer and a supplier.

3.6.3 Heuristics for identifying generalizations

3.6.3.1 Linguistic formulation

As in the other areas, generalizations and specializations can also be identified by specific linguistic formulations.

*"The dog is **a kind** of animal"; "A kind of animal is a dog"; "The boss is a **special** employee"; "**Typical** payment methods are bank transfer or billing".*

3.6.3.2 Uniformity

Generalized classes can be created for classes that have many of the same attributes and possibly also have the same relationships to other classes. This can lead to generalized class names that are not used in the domain.

3.6.4 Recommendations for modeling practice

If all specializations have no attributes, modeling via a property "type" or "kind" is possible.

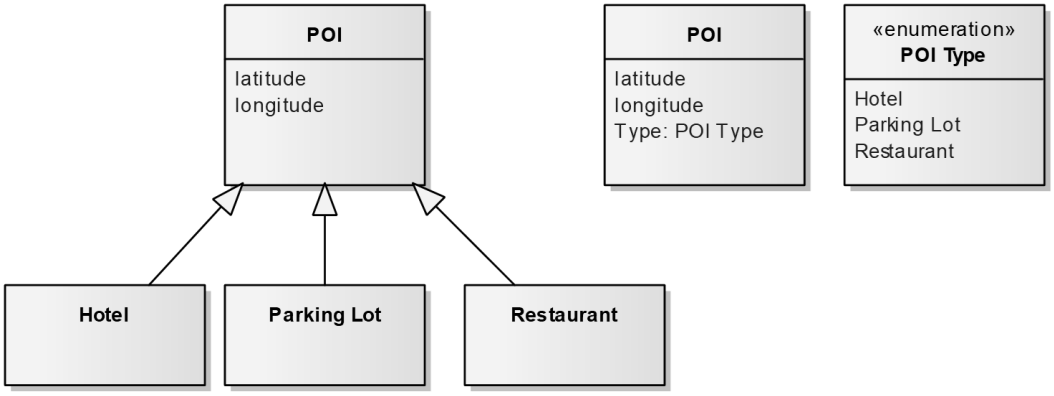


Figure 32: Empty specializations

The choice is determined by the domain experts. If the names of the specializations are anchored in the language of the stakeholders as separate terms, then these should be modeled as independent concepts. If they play a rather subordinate role, an enumeration is preferred.

3.7 Other modeling concepts

3.7.1 Typical concepts and patterns of information structure modeling

In information models, similar structures are encountered again and again. Possible solutions for such structures are called patterns. The main analysis patterns for information models are:

- Item–item description, for example, for a book and specific copy of a book; product and article; invoice and invoice item [CoNM1996]
- Party (also known as a role pattern) [Fowl1996]
- Coordination for, e.g. Processes [Balz2011]
- Composite, e.g., for organization or file system [GaJV1996]

3.7.2 Derived associations

Derived associations are associations that can be derived from existing associations and are therefore redundant. Similar to derived attributes, these associations require a derivation rule. In the simplest case, this is supplemented textually and can simplify the formulation of the requirements because the derivation rule only has to be defined once. An example is shown in Figure 33.

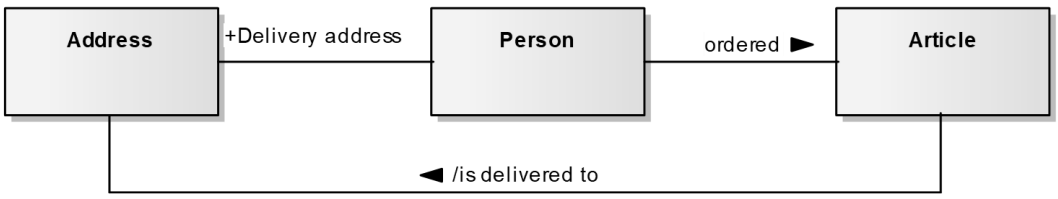


Figure 33: Derived associations

3.7.3 Scope of generalization diagrams

Generalizations can quickly form whole trees with multiple levels. Once such a tree consists of more than 7 ± 2 elements, it should be maintained in a separate diagram.

3.8 Further reading

Creating information models

- Martin, J.: Information Engineering, Book I – Introduction. Prentice Hall, Englewood Cliffs, 1989.
- Shlaer, S.; Mellor, S.: Object-Oriented Systems Analysis – Modeling the World in Data. Prentice Hall, Englewood Cliffs 1988.
- Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley 2005.
- DeMarco, T.: Structured Analysis and System Specification, Yourdon Press, Prentice Hall, 1979.
- Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, Reading, MA 2004.

Analysis patterns for information models

- Coad, P.; North, D.; Mayfield, M.: Object Models: Strategies, Patterns, and Applications, Prentice Hall, 1996.
- Fowler, F.: Analysis Patterns: Reusable Object Models. Addison-Wesley, Reading, MA 1996.
- Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Pattern – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
- Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley 2005.

4 Dynamic views

Program = data + algorithms! With this simple statement, Nicholas Wirth has summarized a complex fact in a memorable way. Applying this equation to requirements, in this chapter we will focus on the desired or required functionality of a system and its behavior (following the description of information models in Chapter 3).

4.1 Dynamic views of requirements modeling

In contrast to the information models, which can essentially be expressed by one diagram type (except for syntactic variants), the dynamic views offer a lot of different abstraction criteria for specifying different aspects of the functionality. This chapter looks at different types of dynamic views in requirements modeling which are summarized in the following table (the last one will be addressed in Chapter 5 of this document).

View	Meaning
Use case view	Decomposition of the functionality of the entire system from a user perspective into processes triggered externally or by time (or interactions or sequences of functions), each leading to a specific added business value for one or more actors in the system context; presented in the form of use case diagrams including textual use case specifications for each use case.
Control flow-oriented view	Specification of sequences of required functions of a system, whereby the emphasis is on the sequence of execution. This view is mainly represented by UML activity diagrams with explanatory activity descriptions.
Data flow-oriented view	Specification of the required functions of a system, including input/output data dependencies; represented classically by data flow diagrams with explanatory descriptions of the functions and data flows between the functions. UML activity diagrams with appropriate extensions can also be used.
State-oriented view	Specification of the event-driven behavior of a system, including states of the system, events, and conditions for state transitions. Represented by state transition diagrams or Statecharts with explanatory descriptions of states, functions, conditions, and events that trigger state transitions.

View	Meaning
Scenario view	Specification of interactions between actors (people, systems) in the system context and the system under development (SuD) that lead to an added business value for one or more actors. Scenario modeling can be done by way of example (e.g., to support the elicitation of requirements) or with a claim to completeness, i.e., all the scenarios which are to be supported by the SuD are modeled (see Chapter 5).

Table 1: Dynamic views in requirements modeling and their meaning

4.2 Use case modeling

Use cases provide a method for systematically describing functions within the defined scope from a user perspective. This section introduces the basic elements of use case models and focuses on a deeper understanding of how to identify and specify use cases.

4.2.1 Purpose

There are many approaches available for breaking the functionality of a whole complex system down into its parts. The approach of breaking down the overall system into processes which provide added value for persons or systems outside of the system has been applied successfully and in many cases (cf. [McPa1984], [JCJO1992], [HaCa1993], [Cohn2002]). A wide variety of concepts and terms is used for such processes, for example EPC (Event-driven process chain), use case, or user story in agile practices.

We consider use case models as a representative of these models. Use case models consist of use case diagrams with associated textual use case specifications. The use case diagrams provide a graphical overview of the required processes of the system and their relationships to actors in the system context. A use case specification specifies each use case in detail by, for example, describing the possible activities of the use case, its processing logic, and preconditions and postconditions of the execution of the use case. The specification of use cases is essentially textual—for example, via use case templates such as recommended in [Cock2000].

The main purpose of use case models is to decompose a complex system into such parts that can be specified afterwards in detail as independently as possible from each other: divide and rule. Since the processes (= use cases) can be derived from the context, this decomposition is neutral with respect to the (existing or planned) inner structure of the system. This means that it does not take into account any internal organizational boundaries or software or hardware limitations of the system under development, focusing instead on the external perspective.

4.2.2 Model elements for use case diagrams

Figure 31 shows the main model elements of use case diagrams, as used in UML. They are used to express the system boundary, actors, use cases, and the relationships between actors and use cases. With regard to the concept of actors, note that actors are always stakeholders in terms of requirements engineering but many stakeholders are not actors because they will never work with the system in operation, even if they want to have a say about the behavior of the system (see [Cock2000]).

Besides the stick figures, various graphical stereotype symbols can be used to express actors. Among others, the use of a clock symbol for time-triggered processes has proven of value, as shown in Figure 32.

Note: by drawing the system boundary, is it easy to distinguish clearly between "inside" and "outside" in use case diagrams. Because of this and since actors are always outside the boundary, it is easy to recognize actors with any kind of representation even without the stereotype << actor >>. Many modeling tools allow you to display or hide the stereotype names like << actor >>. Figure 33 makes use of that simplified notation.

Notation	Name	Meaning
Name	System boundary	The rectangle depicts the scope of the system. Actors are outside the scope. Use cases are inside the scope.
Name «actor» Name *(Alternative)*	Actor	An actor can be a person, a company or organization, or a software or system element (hardware, software or both).
Name	Use case	Functionality of the system, needed by an actor that provides value to the actor. The name should contain a verb, as it describes a functionality, and an object, to which the functionality refers, e.g., "monitor velocity".
Name	Association	The (unnamed) line between the actor and the use case indicates that this actor interacts with this use case.

Figure 34: Model elements of use case diagrams

On the right-hand side, Figure 35 shows an example of a use case diagram with these four basic elements—the system boundary (scope), actors, use cases, and associations.

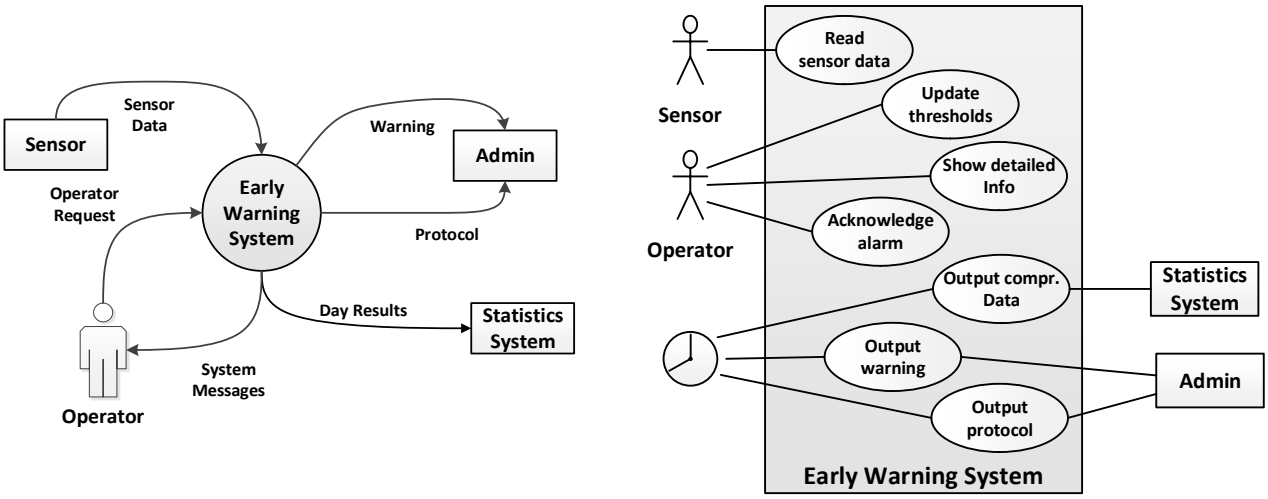


Figure 35: Example of a context diagram (left) and the corresponding use case diagram (right)

4.2.3 Use case diagrams and context diagrams

These two diagram types have similar content but different priorities. Both define a name for the system under development and the system boundary (i.e., the distinction between the scope and context) but with different precision.

The focus of the context diagram is the precise functional definition of the interfaces to *all* neighboring systems. Good context diagrams contain (in addition to the system as a black box) *all* neighboring systems (people, IT systems, devices) that act as a source or sink for information of the system under development.

If a context diagram exists in which all neighboring systems and actors of the system under development are shown, it may be sufficient to create a use case diagram that only contains actors which trigger the execution of use cases.

These actors are called process-triggering actors; they justify the existence of use cases. In other words, without the respective actor there would be no demand for this use case. Therefore, if a context diagram exists, further actors that are involved in the use case (i.e., during the execution of the process after the trigger by an actor) are not necessarily drawn in the use case diagram. This would only increase the complexity of the use case diagram and detract attention from the fact that the use case view mainly serves to decompose the overall functionality of a system into disjoint processes from a user perspective.

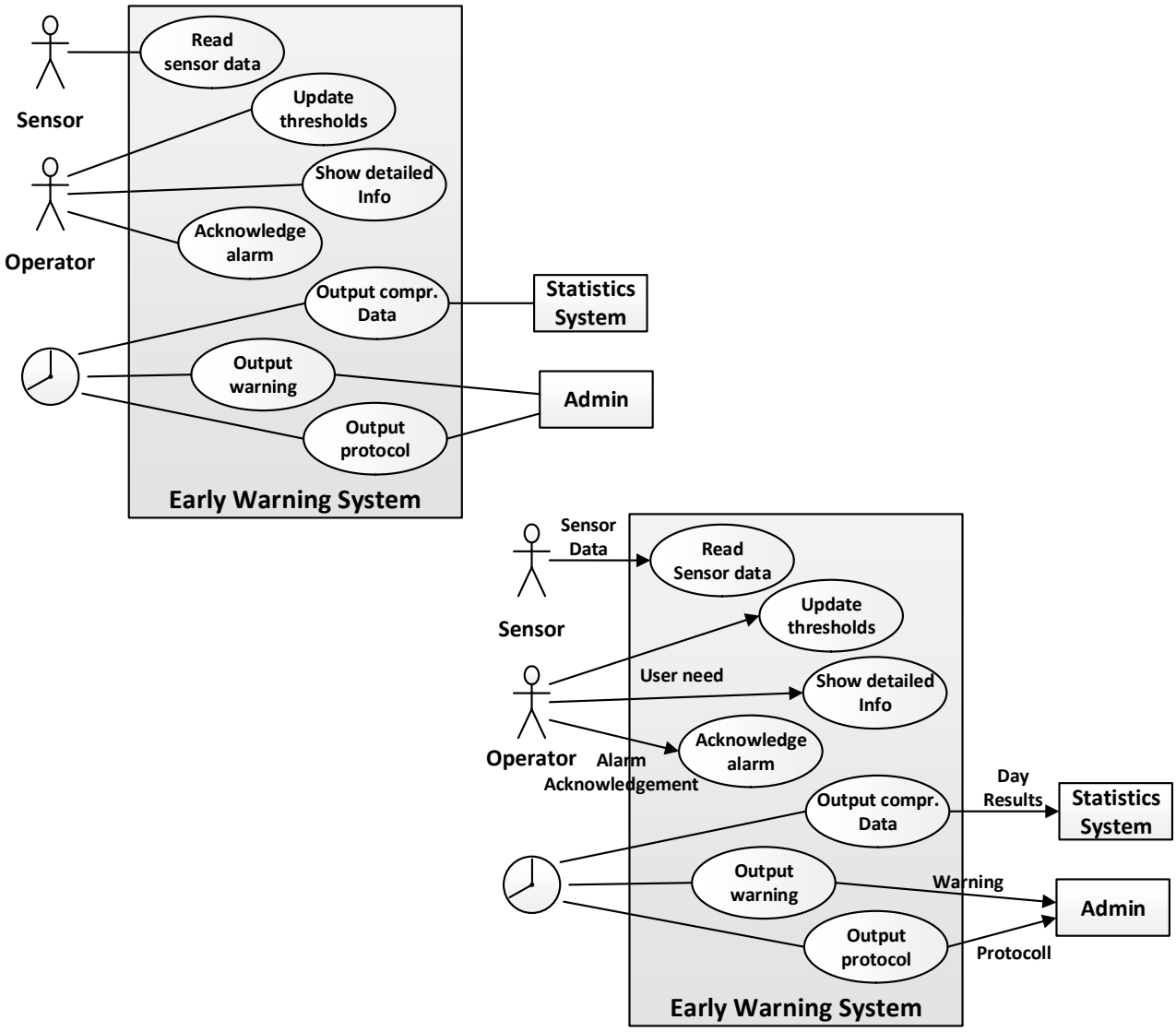


Figure 36: (a) Use case diagram with all neighboring systems, (b) Use case diagram with inputs and outputs

Recommendation 1: Use the strength of both diagram types to obtain on one hand an interface description that is as complete as possible (using the context diagram), and on the other hand, to achieve a rough outline of the functionality from a user perspective (in the form of use cases) that provides a good overview of the required overall functionality and allows a separate, additional specification of each use case.

Recommendation 2: If you only model use case diagrams without a context diagram (e.g., because the tool used does not support explicit context diagrams and the context diagram should not be expressed with a UML class diagram), then all neighboring systems of the system should be included in the use case diagrams. The additional use of graphical layout options allows an easy distinction between actors triggering use cases and other affected neighboring systems (e.g., by arranging the actors on the left and the other neighboring systems on the right). However, such an "extended use case diagram" still does not have the expressive power and precision of a context diagram because in the use case diagrams, the identifiers of the inputs and outputs are missing.

These could be written next to the directed associations between actors and use cases (see Figure 36, b). If we do this, however, the diagram becomes overcrowded and is more difficult to understand. This weakens the major purpose of the use case model.

4.2.4 Finding use cases

In order to find the relevant use cases of the system, it is often useful to focus first on the triggers for possible use cases. Triggers of use cases are events in the system context to which the system under development should adequately respond by executing a process which provides added business value to one or more actors in the system context. [McPa1984] divides these triggers into two categories:

- **External triggers**: An actor (e.g., a neighboring system) wants to trigger a process in our system. Our system will notice this when data coming from the neighboring system crosses the system boundary.

For example, "A guest wants a room in a hotel system". Once the request is received (i.e., the corresponding event in the system context happens), the hotel system should offer a suitable room to the guest.

- **Time triggers**: It is time to execute a process in our system, for example, at specific times or on specific calendar days. By using time events to start a process, there is no need for data to cross the system boundary. It is only necessary that the specified point in time is reached.

For example, in the hotel system: "It is 6pm and thus time to cancel all no-shows and make the rooms available for sale again." Monitoring of internal system resources is also considered as a time event, for example "It is time to reprint our hotel catalog."

4.2.4.1 Continuity of processes from system boundary to system boundary

Each use case should be modeled in a way that the process—once triggered—is considered until its end. The process of a use case should not be interrupted within the system (e.g., at already known software component or organizational boundaries within the system).

The granularity of a use case is therefore determined by the **complete reaction** of the system under development to the trigger from the system context, that is, the primary actors get their added business value after the complete execution of a use case.

4.2.4.2 Pragmatic rules for the granularity of use cases: the 80-20 rule

During use case modeling, the question of adequate granularity for use cases is often raised. In which situations should different use cases be merged into one use case? A strong indication for merging use cases is the criterion regarding whether all processes provide the same added business value.

In large and complex systems, it makes sense to analyze the various use cases. In the case of two use cases having 80% identical processes and similar added business values (e.g., when the processes are nearly identical but are executed with different data), only one use case should be modeled for both processes and the differences between the processes should be documented in the use case specification (see Section 4.2.5).

However, in the case of two use cases having only 20% in common or if many different process steps are needed in the use case description, then separate use cases should be modeled. In the case of a "similarity" of 50%, a decision is often difficult. Ultimately, the similar added business value should be the determining factor for the decision about whether to merge multiple use cases.

4.2.5 Specifying use cases

The popularity of use cases can be explained by the fact that Ivar Jacobson has given the natural language back to the stakeholders for talking about their requirements. He proposed describing the desired process of a use case in natural language. UML does not make any suggestions about the style of use case descriptions. Over the years, many proposals have been made to resolve the weaknesses of purely natural language process descriptions. In particular, [Cock2000] suggests different levels of abstraction of use case descriptions for different groups of readers.

The textual specification of a use case should document the essential inputs and outputs (i.e., data, see also Chapter 3) which are intentionally not shown in the use case diagram.

Detailed textual use case specifications should also describe at least the main flow of control and, if applicable, alternative paths from the perspective of the primary actor (i.e., main and alternative scenarios, see also Section 5.2). Furthermore, they should also specify preconditions and postconditions of the use case execution, which can typically be characterized by states and state transitions (see Section 0). In addition, possible exception events and associated exception scenarios should be documented (see also Section 5.2). Table 2 shows an example of a template for the detailed textual specification of a use case.

Section	Content
ID	Unique identifier of the use case in the development project or program
Name	Name of the use case in the model (this name is shown in the use case diagram)
Trigger	Event that triggers the execution of the use case
Preconditions	Preconditions that must be fulfilled before execution of the use case
Postconditions	Set of postconditions that are fulfilled after successful execution of the use case
Input data	Input data of the use case
Output data	Output data of the use case
Result	Result of the use case, i.e., the added business value which is provided to the actors after execution of the use case
Primary actor	Actor who receives the significant part of the added value of the use case
Further actors	Actors who are involved in the execution of the use case
Main scenario	Normal sequence of activities (execution flow in 70% of all cases, for example). See also Section 5.5.1.
Alternative scenarios	Set of alternative activities. Each alternative process also leads to a successful execution of the use case (e.g., in 30% of cases). See also Section 5.5.2.1.
Exception scenarios	Set of exception scenarios. These scenarios are executed when an exceptional situation occurs in the use case process. These scenarios ensure a controlled error and exception handling. See also Section 5.5.2.5.

Table 2: Example of a template for textual specification of use cases

4.2.6 Structuring Use Cases

UML provides three additional means of expression for structuring the use cases of a system. Figure 37 shows the notations for these three UML elements and briefly outlines their meaning.

Recommendation: Although these structuring elements do exist in the syntax of UML, you should use them very carefully and not too often. Avoiding too many includes, extends, and generalizations keeps the use case diagrams easy to understand and serving their purpose. More complex relationships between use cases can often be expressed in a more understandable and more precise way by using other diagram types, such as activity diagrams (see Section 4.3.3). Both the inclusion of sub-processes (using "Include") and the condition-dependent extension of use cases by sub-processes (using "Extend") can be expressed more precisely in activity diagrams.

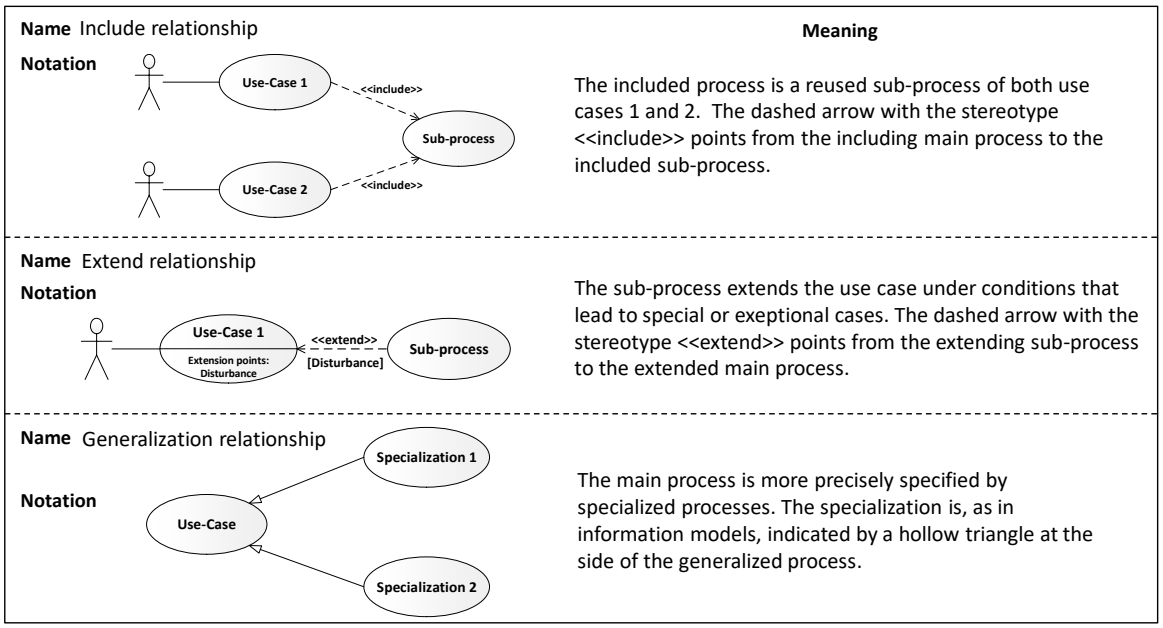


Figure 37: Model elements for structuring use cases in use case diagrams

When applying the model elements mentioned above to structure use cases of a system, the following rules of thumb should be considered:

- An **include** relationship can be used, for example, to explicitly document that several use cases have an identical sub-process. Among other benefits, this saves extra work during specification. Identical sub-processes can also be expressed by using activities with the same name in the activity diagrams which document the process of a use case. Doing this means that there are no additional elements in the use case diagram. The use case diagram remains clear and legible.
- An **extend** relationship can be used to document that an additional sub-process must be executed within the "normal" process of a use case under a certain condition. It is important that the extension point, that is, the condition under which the sub-process is executed in addition, is formulated as precisely and understandably as possible. Since this is often only possible in the use case specification (or in the corresponding activity diagram), it is useful not to model such an extension explicitly in use case diagrams.

- By **generalizing** (or **specializing**) use cases, we can express that specific processes of one or more use cases can be generalized. In most cases, such relationships are modeled when a use case diagram has multiple use cases whose specific processes can be abstracted to a more general level. Figure 37 shows how to model a generalization.
  Experience shows that generalizations are rarely used in use case diagrams since this form of abstraction is rather a concept of information structure modeling in which, for example, common attributes are abstracted by the creation of superclasses (see Section 3). The description of more abstract (generalized) processes compared to their specific (specialized) forms is usually difficult in the context of use case modeling. This model element should therefore only be used after careful consideration and with very specific intentions.

4.2.7 Packaging use cases

For systems with a large number of use cases, it is possible to increase the readability of the use case model by using the following methods:

- Group the use cases according to their business subject
- Create a use case diagram for each group
- Locate the use cases of a group in the same part of the use case diagram

In UML, it is possible to package use cases (similar to packaging other elements of UML). The criteria for packaging can be chosen freely. Usually, logically related use cases (e.g., use cases with a similar added business value) or use cases relating to the same topic (e.g., all use cases for warehouse management in an ERP system) are packaged. Packaging is mainly used to improve handling and readability of a use case model with a large number of use cases.

4.2.8 Summary

Use case models are usually a first step in systematically understanding and specifying the overall complexity of a system (from the context diagram). A textual use case specification is associated with each use case. This specification is usually sufficient to describe the required functionality for simple processes.

For complex processes, this specification is the starting point for the creation of more detailed diagrams that document the required behavior of the system precisely.

The corresponding diagram types are presented in the next sections.

4.3 Data flow-oriented and control flow-oriented modeling of requirements

The core elements of the models from the dynamic view are the functions which should be provided by the respective system. We identified these elements in the **context diagram** and/or in the **use case diagrams** and subsequently specified them initially on a high level.

We will now specify the elements in a more detailed and more precise way by using UML activity diagrams and data flow diagrams (as used, for example, in the Structured Analysis approach [DeMa1979]). Both diagram types will be introduced in this chapter.

The notation element for functions is (historically) different in the two diagram types (see Figure 38) but the purpose of the two diagram types in requirements engineering is the same: a decomposition of the required functionality into smaller functions and the description of the interactions between the smaller functions to provide the functionality required on the higher level.

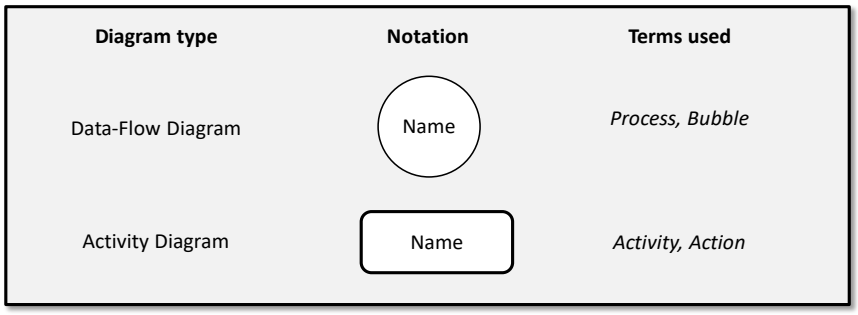


Figure 38: Modeling of functions in data flow and activity diagrams

There are two basic concepts for the interaction of functions – data and control flow, which are motivated and explained in the next section. Here, the two perspectives for "control flow thinking" (here: UML activity diagrams) and "data flow thinking" (here: data flow diagrams) will be considered in more detail.

4.3.1 Purpose/historic overview

One of the earliest models in IT is the flow chart (e.g., according to DIN 66001). Flow charts were used to create program flow diagrams to visualize program logic (at code level). They showed functions (as boxes), alternatives and branches (as rhombuses), and jumps (with anchor links). These diagrams supported programmers in understanding the structure of large programs.

There are two basic approaches for specifying functions and their related interactions further: data flow and control flow. Each of these approaches focuses on different aspects and the approaches are justified and explained in this section. This Handbook describes only one representative for each approach: UML activity diagrams for the "control flow thinking" and data flow diagrams for the "data flow thinking."

Figure 39: Control flow between functions

In the late 1970s, books and publications on "Structured Analysis" [GaSa1977, DeMa1979, RoSc1977] were published. At this point, the focus of analysis approaches changed from considering the control flow to modeling the data flow. Data flow diagrams also examine the functions of the system (usually represented as circles, in some notations as rectangles with rounded corners, or as rectangles).

Nevertheless, the (labeled) pointers between the function blocks have another meaning. The pointers in the data flow diagrams represent inputs and outputs of functions, that is, the **data flow** between the functions and not the control flow (see Figure 40).

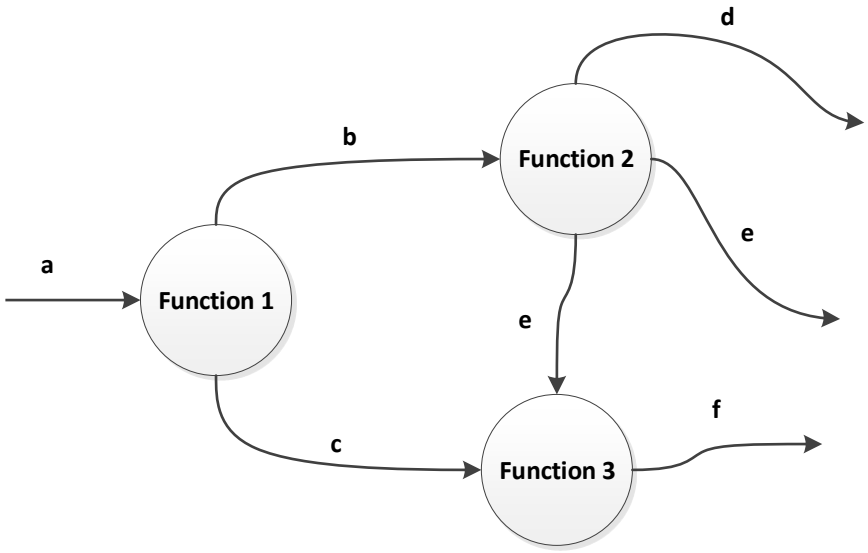


Figure 40: Data flow between functions

In data flow-oriented views, **all** functions can be active simultaneously. The data flow specifies only causal dependencies, meaning that a function can only work when its inputs are available. However, in contrast to a control flow, no explicit sequence of the functions is modeled.

With the introduction of UML in the late 1990s, the emphasis on control flow based on activity diagrams was introduced again. UML activity diagrams are very suitable for modeling process flows. They visualize the control flow between activities or actions of the system. If the sequence of activities is sequential, the follow-on action can only start when the preceding action is completed. Alternative control flows can be expressed using decision points. Concurrent control flows can also be expressed.

In activity diagrams, functions are represented by boxes, control flows by arrows, and decision points by diamonds.

To summarize: complex required functionality can be modeled either in a control flow-oriented way (by using activity diagrams) or in a data flow-oriented way (by using data flow diagrams). We should focus not on the choice between the two diagram types but rather on the fundamental thinking in data flows or in control flows. Both concepts are useful and as explained below, you can also represent data flow thinking in UML activity diagrams and conversely, express relatively linear processes with data flow diagrams.

Note: in some modeling approaches of the dynamic view, such as in Petri nets, the proposal is to model the data flow and control flow together in the diagrams. This often leads to a higher complexity in the diagrams, making them difficult to understand.

4.3.2 Requirements modeling with data flow diagrams (DFDs)

Data flow diagrams are often used to model requirements from a data flow-oriented perspective. They model the functionality of the system under development using functions, data stores, data/information flows, as well as sources and sinks.

4.3.2.1 Model elements of data flow diagrams

Figure 41 summarizes the main model elements of data flow diagrams.

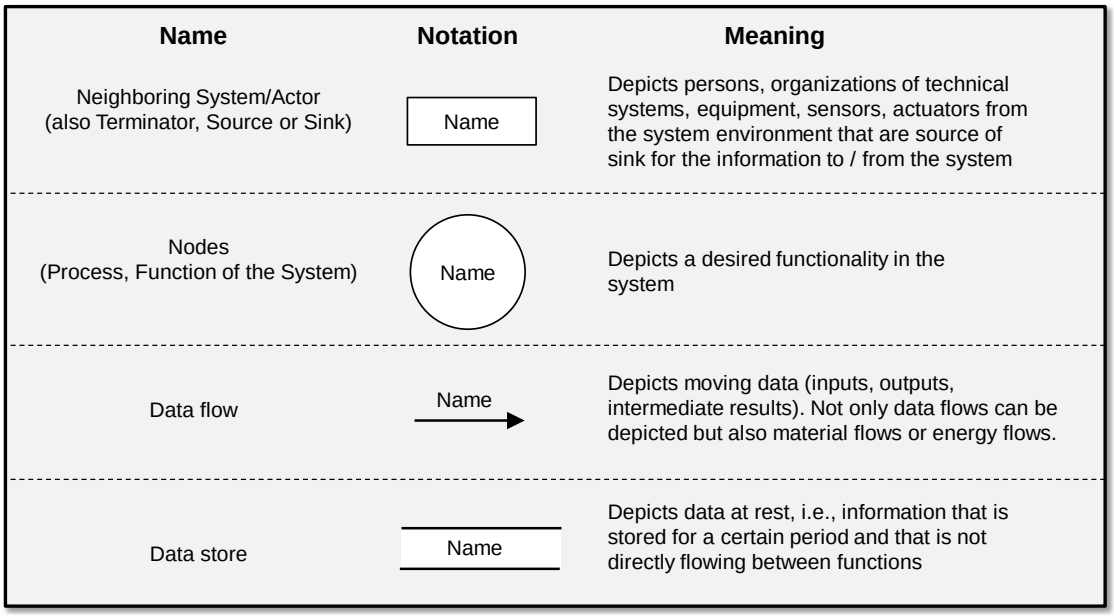


Figure 41: Model elements of data flow diagrams

Figure 42 shows an example of a navigation system using the four elements that can be used in data flow diagrams. It also provides further information on the semantics.

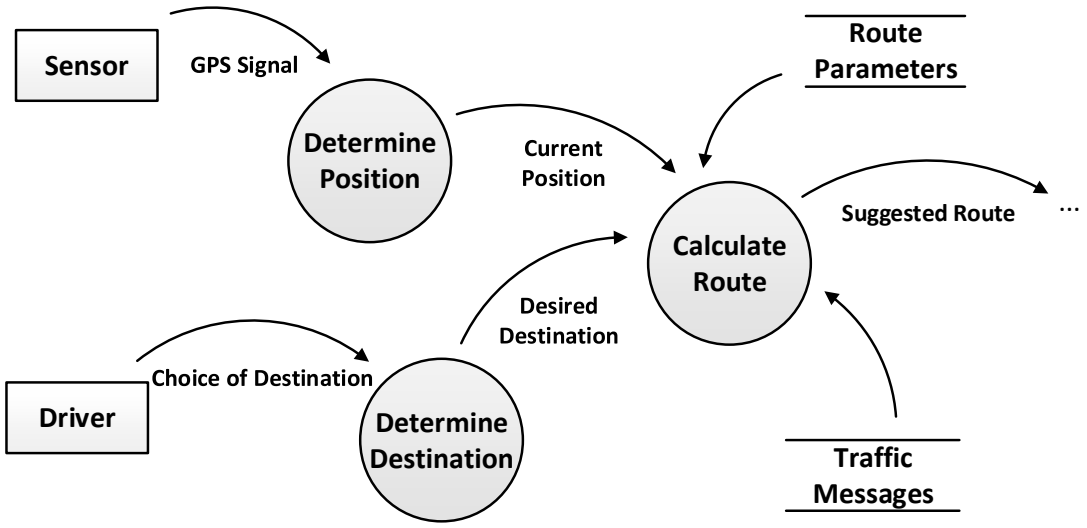


Figure 42: Example of a data flow diagram (part)

Data flows (such as GPS signal or desired destination) represent data in motion.

Data stores (such as route parameters, traffic messages) represent data at rest. Data in data stores can be created and updated by one set of functions and read (non-destructively) by another set of functions. It is persistent data. The period for which the data is to be stored is not specified.

The fourth element (the rectangles, in the example "sensor" and "driver") represents neighboring systems of the system under development. In the Structured Analysis approach, they are called **terminators** or **sources** and **sinks**, depending on whether they provide inputs or receive outputs. A terminator may be both a source and a sink.

These terminators are usually listed completely in a context diagram (see Section 2.2). From this perspective, the classical context diagram is a specific data flow diagram in which **all** neighboring systems (or actors) and all input and all output data are modeled; however, the functionality of the system under development is compressed into **a single** node. If the neighboring systems (or actors) are already shown in the context diagram, then in the refined data flow diagrams, often no terminators are shown and only the associated data flows at the system boundary are modeled (see Section 4.3.6).

For data flow diagrams, the following fundamental rule is valid: All input and all output data must be shown in the diagram.

The data flow specifies causal dependencies, which means that a function can only work when its inputs are available. However, in contrast to a control flow, no explicit sequence of the functions is modeled.

If there is a need to express the sequence of functions **explicitly**, data flow diagrams can be supplemented by state transition diagrams. State transition diagrams use events and states to express the sequence of functions.

The collaboration between data flow diagrams and state transition diagrams can be illustrated by the metaphor of a string puppet or marionette. The functions in the data flow diagram correspond to parts of the puppet (such as arms, legs, head) which can be moved freely and relatively independently of each other. A state machine corresponds to the wooden cross with the strings to the moving puppet parts. The wooden cross makes a (moving) connection between the moving parts of the puppet, whereby the puppetry can restrict the possible movements of the puppet parts.

4.3.2.2 The relationship between data flow modeling and use cases, control flow modeling, and information structure modeling

The data flow-oriented modeling of requirements using data flow diagrams has a substantial connection with the context diagram, the use case view, and the information structure view. Use cases are a tool for systematically specifying the functions within a defined scope from the user perspective and at a high level. During requirements engineering activities, these functions need to be detailed and decomposed into more detailed system functions and their dependencies.

The system functions of a use case, including data dependencies between the functions and with actors (terminators), can be modeled using data flow diagrams. The more detailed system functions can be identified during the functional analysis of the use case scenarios (see also Section 5.5.3). The structure of the data, which is modeled in the data flow diagrams as data flows ("data in motion") and as a data store ("data at rest"), is defined in the diagrams of the information structure view (see Section 3.1).

4.3.3 Requirements modeling with activity diagrams (ADs)

UML activity diagrams can be used to model requirements from the control flow perspective. Activity diagrams specify the required processing logic of use cases, system functions, or processes that need to be delivered by the system under development so that it fulfills its purpose during operation.

4.3.3.1 Model elements of activity diagrams

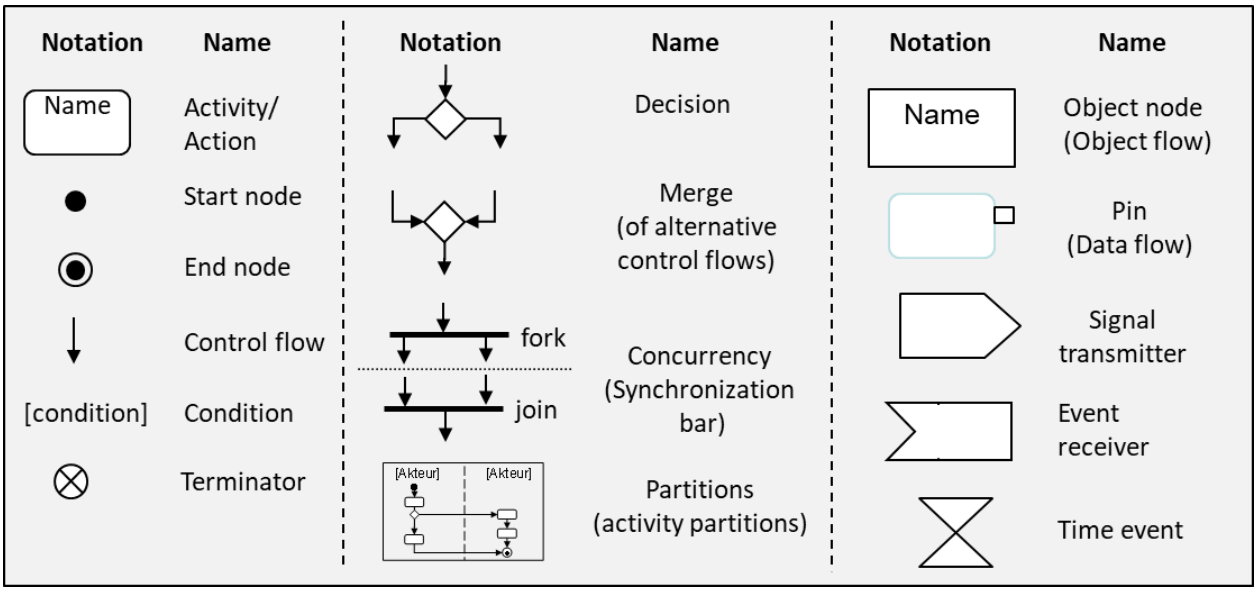


Figure 43 summarizes the main model elements of activity diagrams.

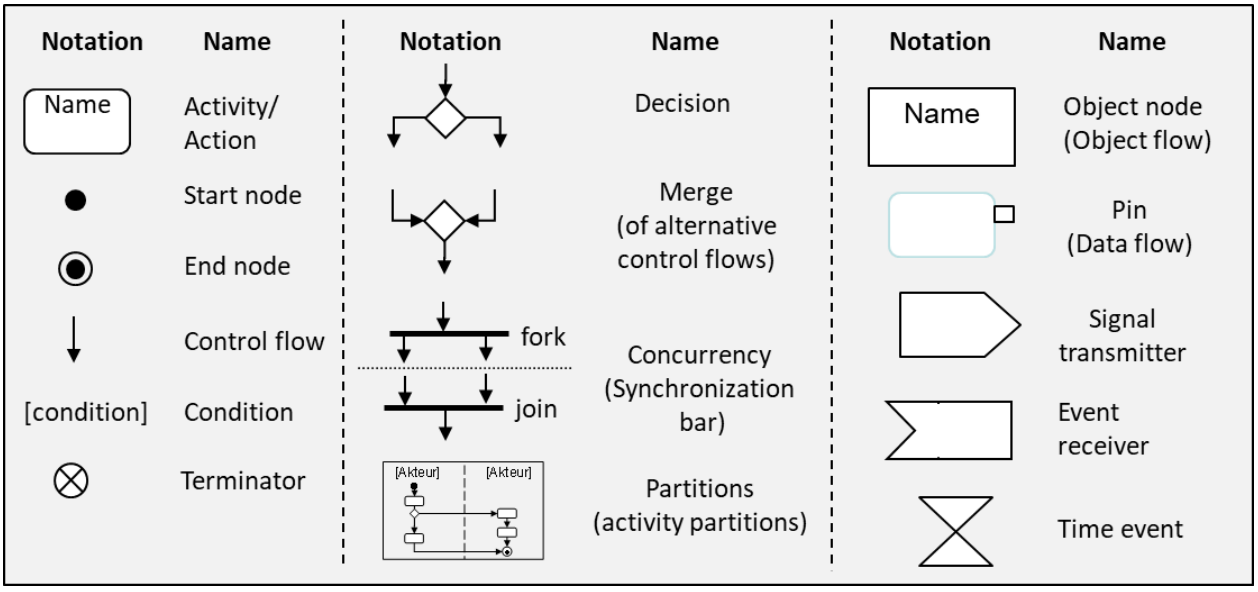


Figure 43: Model elements of activity diagrams

Activity diagrams document the control flow between activities or functions of the system. The control flow starts at the start node and ends at the end node(s). The diagrams can be used to model sequential processes, branches of the control flow (using decision points), and concurrent processes (using synchronization bars). Concurrent processes contain activities which can be processed independently and therefore potentially at the same time.

They are particularly important for the system analysis because in real systems, many things can happen simultaneously or independently of each other and not strictly sequential.

For the exact syntax and semantics of the notation elements, please refer to advanced books on UML, such as [RuJB2004, BoRJ2005]. Figure 44 illustrates the use of the typical

model elements of activity diagrams and the essential syntactic rules with an abstract example.

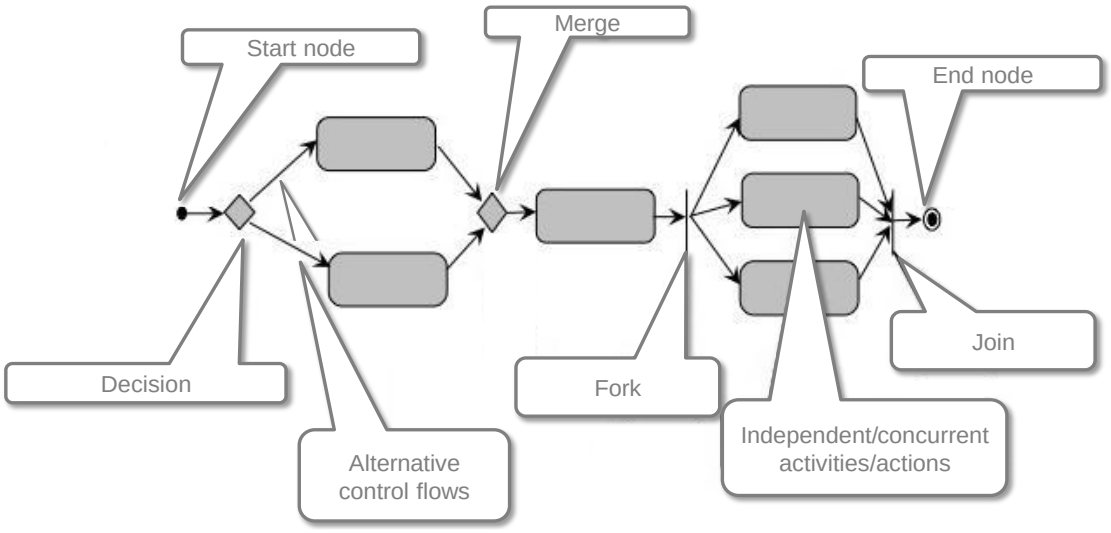


Figure 44: Using the model elements of activity diagrams

4.3.3.2 Modeling object and data flows in activity diagrams and their relationship to information structure modeling

Activity diagrams also allow us to model object or data flows, as shown in Figure 45 and Figure 46. This is done by inserting objects (see Figure 45) or parameters of the activities (see Figure 46), as well as all accesses to data stores, are included in the diagram. In contrast, activity diagrams do not define how much or how little data is displayed in the diagrams.

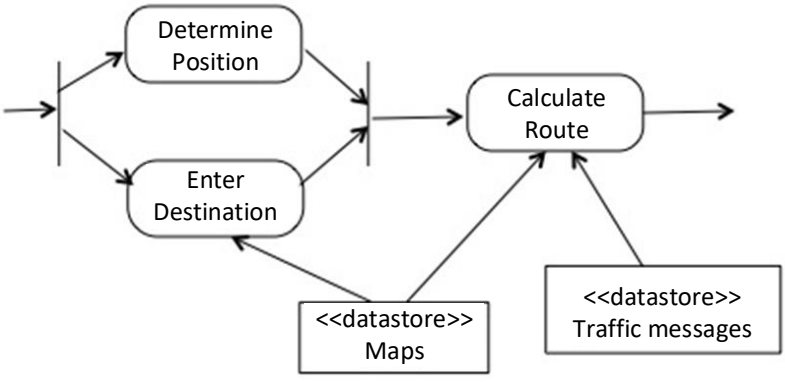


Figure 45: Modeling object flows in activity diagrams

The example in Figure 45 shows that the activity "Calculate Route" requires an input from the objects "Maps" and "Traffic messages". However, it does not show the main output (the

route or several route suggestions). It also does not show any route parameters used (such as "fastest route", "shortest route").

In contrast to data flow diagrams, where extreme importance is placed on the completeness and consistency of the models, UML diagrams are supposed to be "useful" mainly for the communication between the persons involved. The completeness of the specification can be achieved with supplementary activity descriptions.

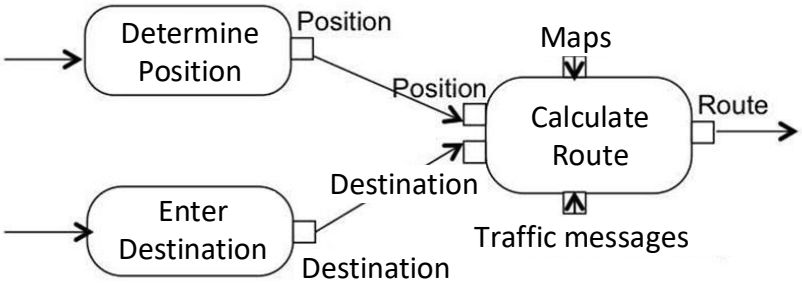


Figure 46: Modeling flows in activity diagrams using pins

The "pins" at the functions represent the inputs and outputs of the function. Thus, relationships, such as that "Determine Position" creates a "Position" as output and "Calculate Route" requires a "position" as input, can be represented graphically.

By using activity diagrams, the modeler can choose to include no data (objects) in the diagram or to intentionally add some data (objects) to highlight certain aspects. It is important to note that all inputs and outputs must be fully specified in the requirements specification (at the latest in a textual specification of each function, see Section 4.3.5). The structure of data or classes and their dependencies to each other should be modeled in the information structure view (see Section 3.1).

4.3.3.3 Relationship of activity diagrams to use case and scenario modeling

Activity diagrams are often used to specify the processing logic of use case scenarios in detail (see Section 4.2.5). Activity diagrams are created to visualize the scenarios, which are processes with activities and processing logic. As long as the diagram remains understandable, the main scenario can be modeled jointly with the alternative scenarios and the exception scenarios as part of the same diagram.

This is typically done by using decision points, where the control flow branches. Depending on a condition, either the process logic of the main scenario or the process of the alternative flow/exceptional flow is executed.

Figure 47 with an example of a control flow related to a use case. There are many decision points where it is possible to switch between the scenarios. In this example, there is one switching point before the activities "Enter destination address via keyboard" and "Say

destination address". These activities belong to different scenarios. Exceptional scenarios can be modeled using decision points. Figure 47 shows this at the last decision point. It defines that in the case of the exception "Map information not available" the activities "Issue error message" and "Shut down system" are performed.

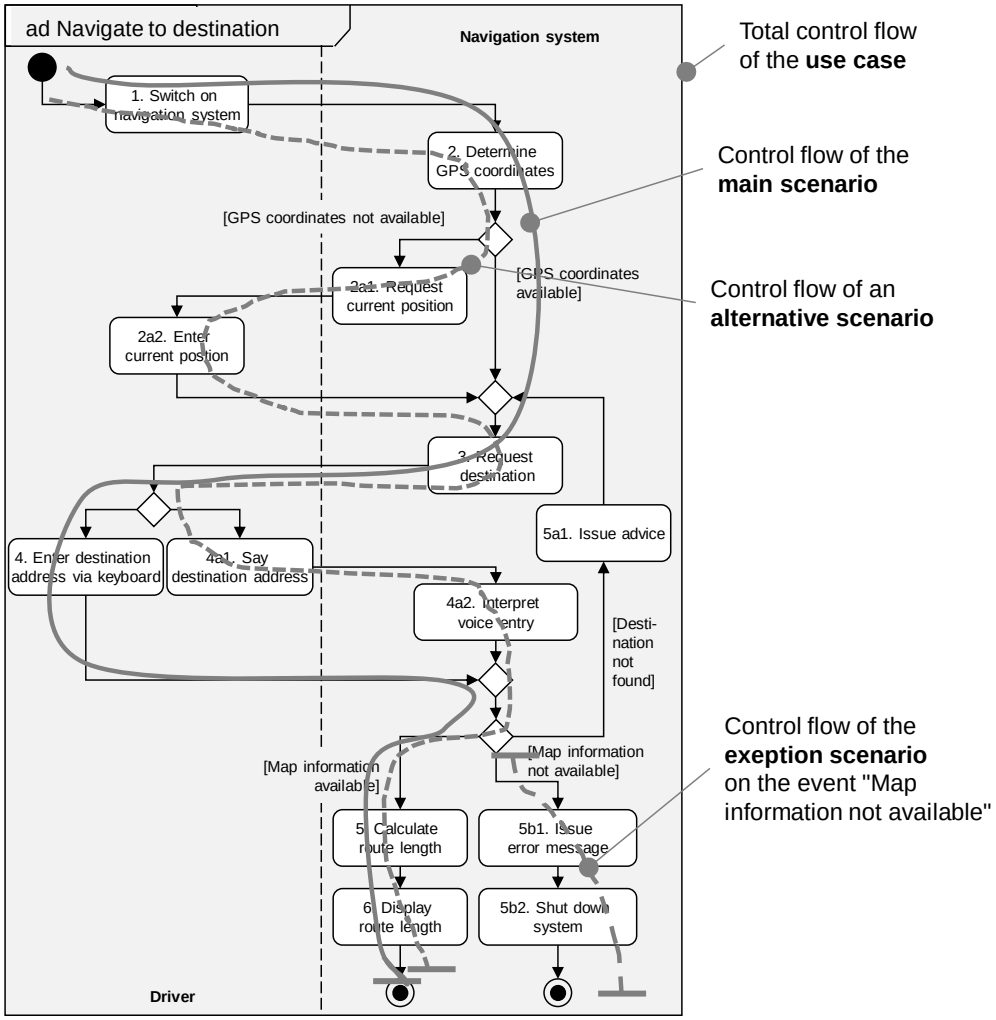


Figure 47: Modeling the control flow of use cases using activity diagrams

To model exceptions which do not appear at a specific location in the control flow but in an area of the control flow or during execution of the whole use case, signal inputs and outputs and interruptible regions may be used (see Section 4.3.7).

For all UML diagrams, it is important that they are easy and understandable. In this case, they should visualize the processing logic of a use case in a way that allows the reader to easily recall the context. The recommendation is therefore to show only a few aspects (scenarios) in one diagram. Further aspects (scenarios) can be shown in additional diagrams. It is also possible to create a diagram with the main scenario and further diagrams for each alternative scenario together with the main scenario. The textual description may contain further details.

4.3.4 Decomposing or combining functions

Both types of diagrams (data flow diagrams and activity diagrams) support the decomposition of complex functions into simpler functions as well as the combination of simpler functions to form more complex functions. In other words, data flow diagrams and activity diagrams can represent hierarchies of functions (see Figure 48 and Figure 49).

This abstraction mechanism allows us to structure complex issues in order to keep them understandable and manageable. Within the dynamic view of requirements modeling, this hierarchy is a powerful tool for controlling the scope and complexity of the systems under development.

In Figure 48, the complex function "Determine Destination" of a navigation system is decomposed into five steps (which are not specified in the example diagram).

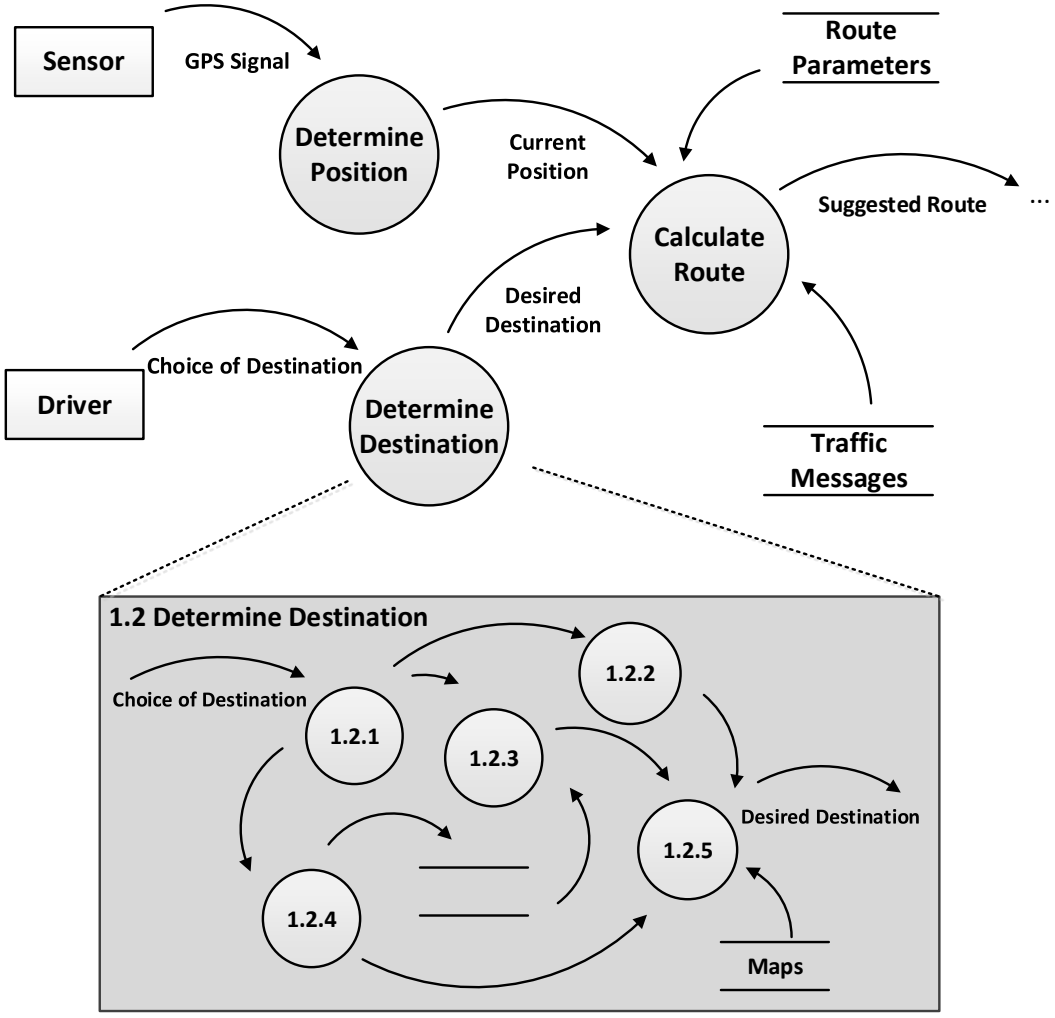


Figure 48: Hierarchical decomposition and combination of functions in DFDs

In Figure 49, the complex activity B is decomposed into a detailed process consisting of five activities. Conversely, the detail activities B1, B2a1, B2a2, B2b, and B3 can be combined to form the aggregated activity B.

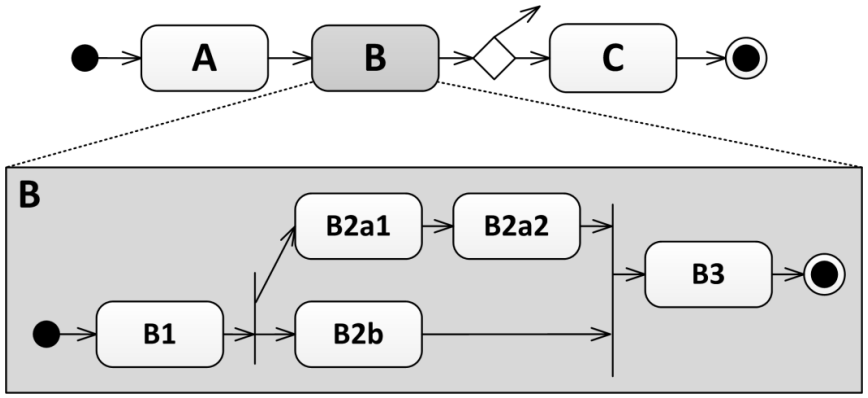


Figure 49: Decomposition of a function in an activity diagram

In addition to content-based criteria (such as a technically strong relationship, which is often manifested in finding a good name for the whole of the detail activities), very pragmatic criteria are applied for decomposition or combination. One criterion is usually that the diagram should fit on one page of a document. Furthermore, most methods recommend modeling no more than 7 ± 2 functions per diagram.

4.3.5 Textual function specifications

How "far" (level of detail) should the functions be decomposed in data flow diagrams or activity diagrams? In other words: when should the decomposition of functions stop? A simple heuristic rule is the length of the required function description. If the precise specification of the requirements of a function needs more than a half-page description, the function should be refined again to avoid natural language specifications that are too large.

If the diagram already expresses everything that needs to be stated, then you have probably decomposed too far. Models are easier to understand and read if you do not model the last one to two decomposition levels and instead, specify the functions in text form (for example, on half a page). It is also possible to refine a function (activity) by assigning a limited number of three to seven simple, natural language requirements which specify the considered function in detail.

Example: Textual description of the function "Determine Destination" (see Figure 48)
Function: determine destination
Input: destination selection (done by the user of the navigation system), map
Output: desired destination

The function should provide the user with four options for selecting a destination:
- By entering an address using the keyboard
- By entering an address using voice entry
- By selecting from a list of stored addresses
- If a map is displayed, by selecting a destination via the touchscreen

For most users of these diagrams, the above-mentioned refinement level with a specification on half a page is sufficient to understand the functional requirements and to systematically derive test cases. This is especially true for testers who need to verify, after completion of the system development, whether the system in operation implements the requirements completely and correctly.

4.3.6 Ensuring consistency between requirements at different abstraction levels

A requirements model contains diagrams and textual specifications at different levels of abstraction (see Section 0). It is important to keep the requirements at the various levels of abstraction consistent with each other. As part of the data flow view, such consistency conditions have been introduced in the form of "balancing rules" (cf. [DeMa1979]).

These **consistency rules between diagrams at various levels of abstraction** can be adopted in the same way for activity diagrams:

- Inputs and outputs of a function at one level must be consistently present as inputs and outputs at the next lower level. This begins with the context diagram as the most abstract representation. Each decomposition of the context diagram must include all interfaces that were already included in the context diagram. The inputs and outputs at the next lower level do not need to have the same name because data can be decomposed, as can the functions. For example, on the higher level, we find the output "statistics" and at the next lower level "product statistics", "regional statistics", and "sales statistics". This decomposition is usually described in a glossary (or data dictionary) or modeled in the information structure view. The ground rule is that the higher level may contain more abstract concepts which are specified more precisely during refinement.
- A special rule applies to the balancing of data stores: data stores should be introduced only at that level of abstraction where they offer an interface between at least two functions. In other words, a data store which is written and read by the same function should be hidden inside the function (i.e., it should be shown only in a refinement of this function). A data store should not be shown in a diagram where it is needed only by one function. From the abstraction level at which the data store is first modeled, the read or write access to this data store must be repeated at each lower level.

Even though activity diagrams usually do not model data flows and data stores, the balancing rules should be considered. The review/verification of requirements must cover both the diagrams and the supplementary descriptions. You have to check that the refinement of diagrams and specifications is consistent at all the different levels.

4.3.7 Interruptible activity region and receiving/sending messages

Using an example, this chapter introduces the last two model elements for activity diagrams which are relevant for requirements engineering: the **interruptible activity region** and the **receiving/sending of messages**:

> **Example:**
> A user should have the option to select a person for whom the account transactions should be displayed. While the transactions are displayed, the user can close the window or select another person. New transactions can also be received by the system. Thus, the content in the window should be updated automatically.

In Figure 50 the desired behavior of the system is modeled using an activity diagram. The box with dashed lines defines the interruptible activity region. All actions that are in the diagram can be interrupted when signals are received (in the example, only the activity "Display account transactions"). If a signal receipt is modeled within the interruptible activity region, all actions in the region will be interrupted when a signal is received.

To better distinguish the signals and to further specify the trigger of the signal, the stereotypes "User action" and "System event" are used. After receipt of a signal (and the interruption of the current action), if necessary, another action is executed and the cycle can start again (here: after receiving the signal "New transactions").

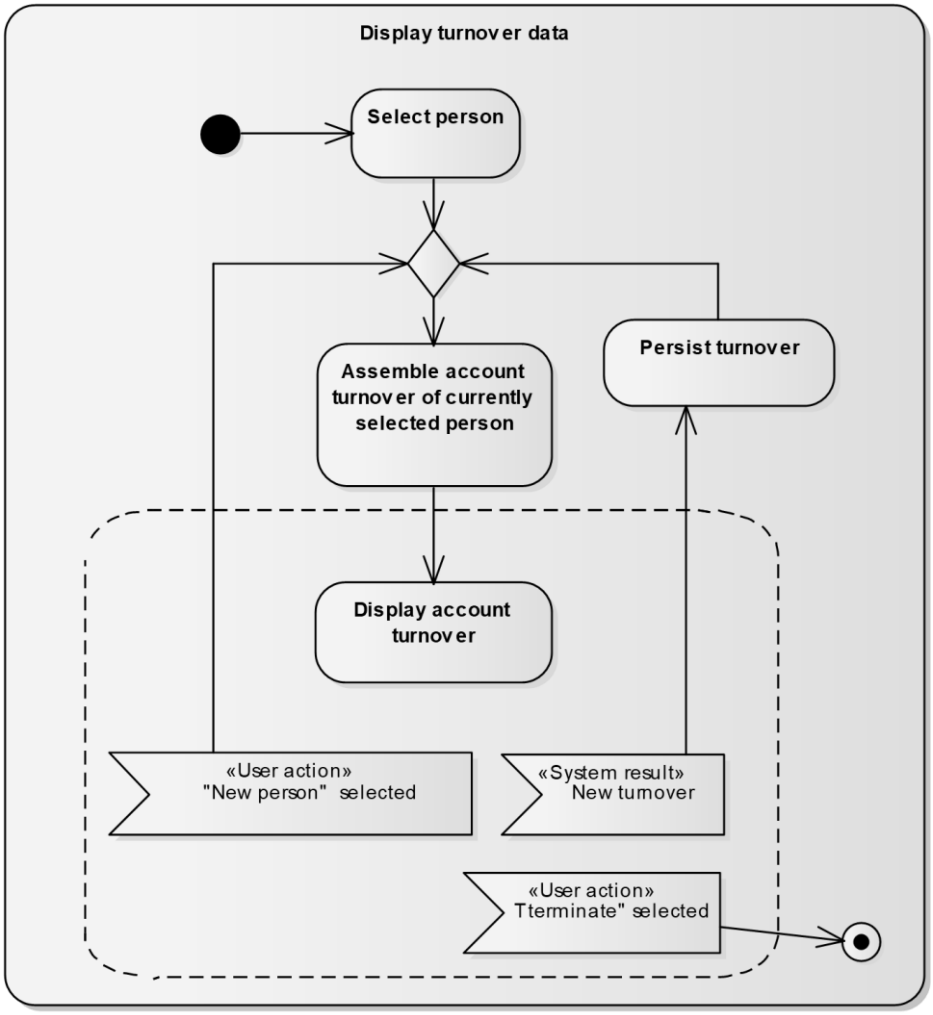


Figure 50: Example of the modeling of signals in an interruptible activity region

The user terminates the activity by clicking on "Cancel". To complete requirements analysis, the activities in this diagram should be further specified by refined activity diagrams or textual specifications. The following must be specified:

- exactly how and in which sequences the transactions are to be displayed
- which options the user has for selecting another person

Signals can also be created and sent (and not only received) as part of an activity diagram. An example activity diagram for a type of function known as heartbeats is provided in Figure 51. A sign of life is sent out every second. This is triggered by a "Time event" (the hourglass), which stops the flow each time for the specified time (one second). Again, an interruptible activity region is used to indicate when the heartbeat should stop.

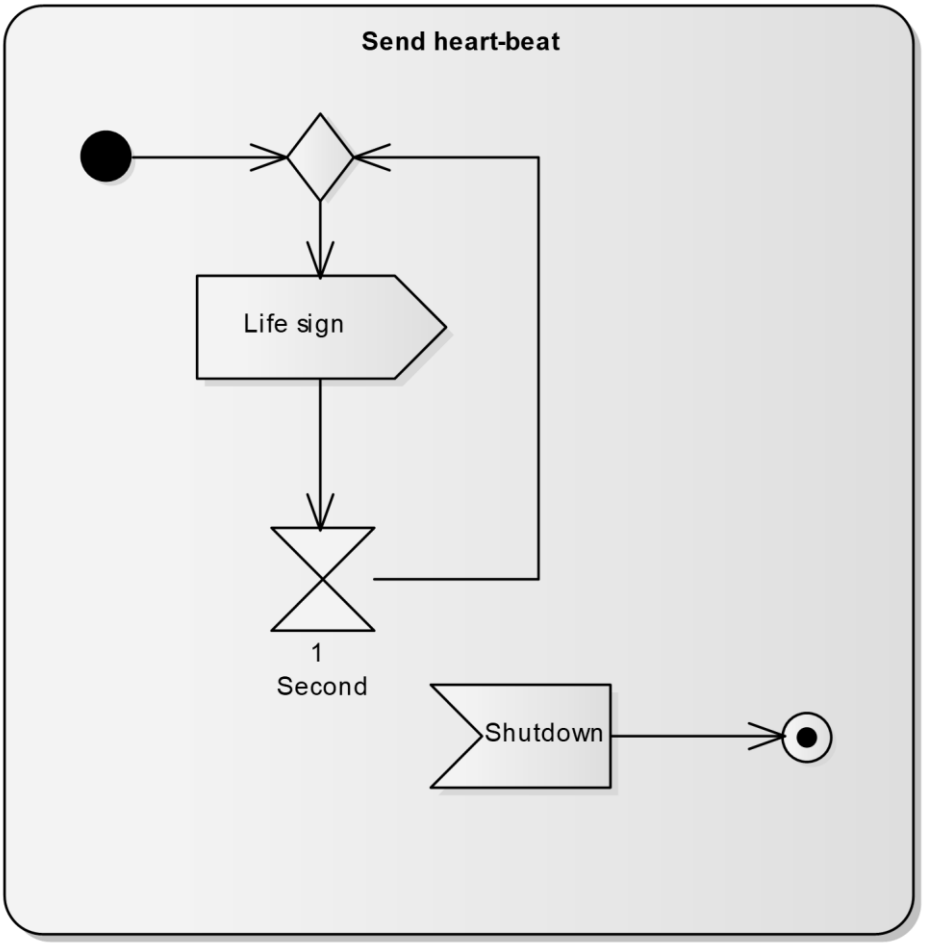


Figure 51: Example of heartbeats

4.3.8 Comparison of data flow diagrams and activity diagrams in requirements modeling

The concepts behind both diagram types and the available model elements have a big influence on our thinking. In activity diagrams it is easy to express: "F1 is executed before F2" (indicated by an arrow). In data flow diagrams, it is easy to express: "F1 produces D as output data and F2 needs D as input data" (with a labeled arrow).

Activity diagrams	Data flow diagrams
Emphasis on control flow (processing logic) ▪ Sequences ▪ Branches after decisions ▪ Concurrency (fork/join)	Emphasis on input/output dependencies (data dependencies) ▪ Who produces what? ▪ Who needs what?
Inputs and outputs have less importance	Control flow (processing logic) has less importance
In the case of sequential activities, the completion of one activity triggers the activation of the next activity	Availability of inputs allows the execution of a function (process)
Strict time flow (apart from concurrent control flows, i.e., fork/join)	No implied sequence (except for the causal dependency induced by data dependencies)

Table 3: Differences in requirements modeling with data flow diagrams and activity diagrams

To summarize: the emphasis in the modeling languages has shifted back and forth over the decades. It started with the emphasis on control flows (in flow charts and program flow diagrams). Later, the emphasis changed to data flows (in DFDs) and back to control flows again (with UML activity diagrams).

Both concepts—control flow and data flow—are useful tools to support thinking, visualization, and specification of required functions and their dependencies. A requirements engineer should be familiar with both concepts and know how they can be used. Due to the current dominant position of UML and the corresponding tools, you will probably use activity diagrams. However, you should be able to deal with data flows and data stores in this notation too.

4.4 State-oriented modeling of requirements

Requirements are mostly derived from dynamic views of the system. The requirements of a system also can be modeled using a state-oriented view, with a finite set of states and associated state transitions. This view is particularly important for systems whose behavior:

- Specifically depends on what has been done already (history)
- Is strongly influenced by asynchronous events

4.4.1 Purpose

State-oriented modeling allows clear specification of preconditions and postconditions. These conditions are required for the execution of a function (e.g., a use case or an activity in the activity diagram). This type of modeling can be applied to the total system or parts of the system. If it is used to model parts of the system, the model can be arranged in a similar way to the use cases distinguished (see Section 4.2).

In addition to modeling the states of a system, state machines can also be used to model the states of a branch-specific object that is described in the information view (see Chapter 3). As a result, the effect that different system functions have on that object is shown in an overview within one state machine. Compared to the purely functional view, for example, in the process-oriented view, a redundancy is introduced which serves one of the following purposes:

- The consistency in the specification of the functions is validated.
- A focused view of an object increases the comprehensibility and traceability.

It is important when dealing with state machines that the topic under consideration (the matter at hand) for which the states are modeled is determined consciously. It may be one of the following:

- The system under development
- Subsystems of the system
- The objects of a class from the information view

4.4.2 The term "state"

The term "state"—as generally used in requirements engineering—is derived from the theory of automata: a state is a summary of certain conditions that apply for an object of observation over a period of time.

But where do the "conditions" for an object come from?

If the item in question is an object (an instance of a class), then the possible states are described by combinations of possible values of its attributes. Figure 52 shows an example of a car with six possible values for the color and two possible values for the attribute "Ready to drive". As a result of these potential conditions, a total of 12 potential states for the car are available.

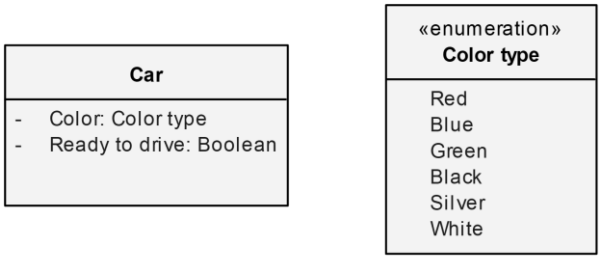


Figure 52: Definition of a car (a)

Extending the example to another attribute that specifies the mileage, we encounter a problem if this attribute can have an infinite number of possible values (see Figure 53). The number of potential states is therefore unlimited, and this can no longer be represented graphically in the form of a finite state machine.

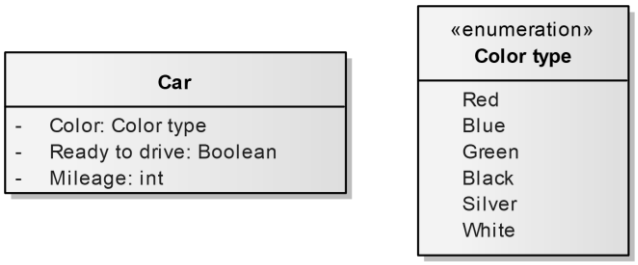


Figure 53: Definition of a car (b)

Methods for reducing the number of states to a manageable level are described in Section 0.

The theory of finite automata (Moore or Mealy automata) is not used widely in requirements engineering. Statecharts, introduced in 1987 by Harel [Hare1987], or the extension of Harel Statecharts in the OMG UML [OMG2010b, OMG2010c] and the OMG SysML [OMG2010a] are used instead.

The Harel Statecharts differ from the original finite state machine mainly regarding the following three points, which greatly simplify the modeling of the state-oriented view of requirements engineering:

- More extensive ways of linking functions to states and state transitions
- Introduction of conditions (guards) which, for example, have to be met before the transition
- Introduction of the possibility of hierarchical state machines and orthogonal regions

The second point in particular has huge implications for modeling the state-oriented view, as it is no longer necessary to model the entire history in the form of conditions. This reduces the number of observed states and the complexity of the charts created.

State machines have one property in common: the object of the state machine is always in a defined state at the moment of observation. This implies that the transition between two states has no temporal aspect (consumes no time).

In a real life implementation, however, for example in software, these transitions do consume time. Therefore, the phrase at the beginning of this paragraph must be expressed a little more softly: an object can respond to events from the outside only if it is in a defined state. With respect to the implementation, this means that the incoming events must be buffered for the short duration of the transition. This ensures the required semantics of a state machine.

4.4.3 A Simple Example

The diagram in Figure 54 contains a simplified state machine for a windshield wiper system in vehicles. In this example, the main model elements for modeling a state-oriented view are presented. They are presented in more detail in the following sections along with the notation elements of UML.

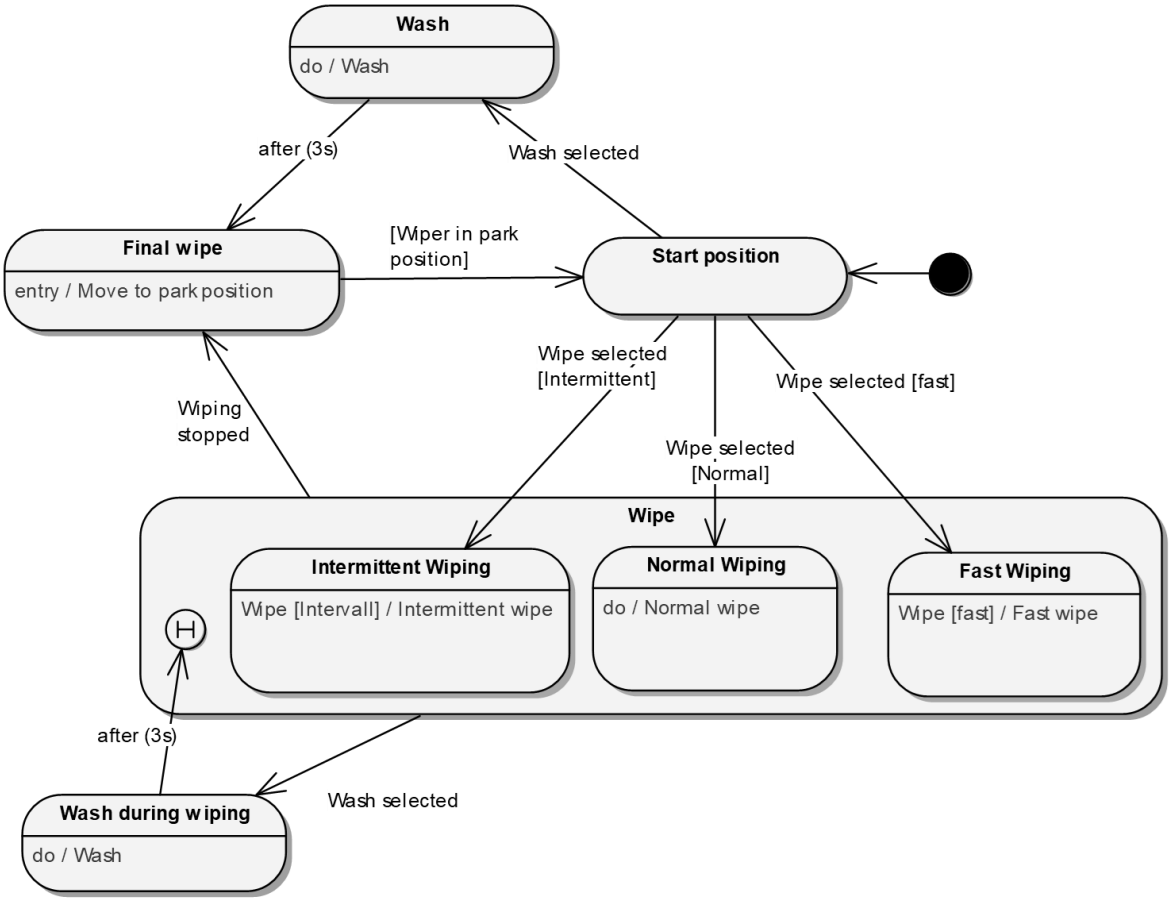


Figure 54: State diagram for a wiper system

4.4.4 Model elements of state machine diagrams

In this section, we present the most commonly used model elements for modeling a state-oriented view. We use the notation of UML. For more notation symbols and explanations, see [OMG2010b, OMG2010c], and [BoRJ2005].

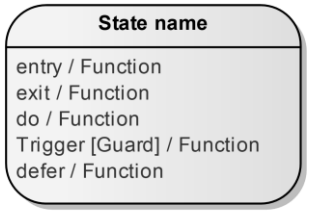
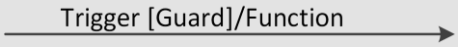
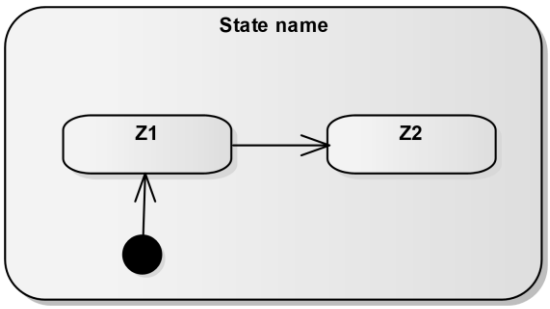
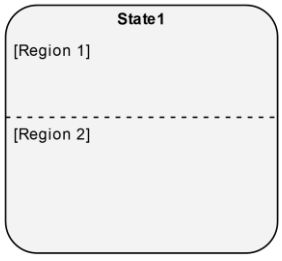
Notation	Name
	Simple state
	Transition
	Initial state
	Final state
	Composite state
	Sub-machine state
	Orthogonal regions

Figure 55: Modeling constructs of state machines (detail)

4.4.4.1 Simple state

Syntax and semantics

In UML, a simple state is represented with the notation element shown in the following figure:

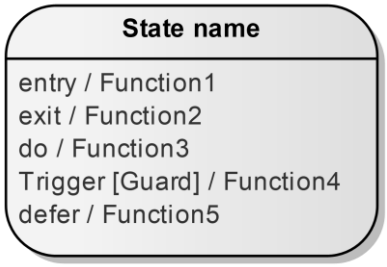


Figure 56: Notation of a state

A state should always have a name. In addition, in this state you can specify which functions are called. In UML, the types of function calls listed below are defined in a state and the *italic* identifiers are defined with keywords with specific semantics. The identifier "function" refers to the function that is executed.

- **Entry behavior:** entry/function: When a state is entered, the function is executed. This function cannot be interrupted.
- **Exit behavior:** exit/function: When a state is exited, the function is executed. This function cannot be interrupted.
- **State function:** do/function: While the object of observation is in the state, the function is executed. This can be interrupted by a trigger which leads to a state change.
- **Triggered function:** trigger [guard]/function: When the trigger occurs and if the guard is true, the function is performed without the object exiting the state.
- **Delay:** trigger [guard]/defer: If an event in the deferred event list of the current state occurs, the event is deferred for future processing until a state is entered that does not list the event in its deferred event list (see Section 4.4.4.2)

For the states, the following rules apply:

- A state is entered when a transition is passed through that leads to this state as the end point (see Section 4.4.4.2).
- A state is exited when a transition is passed through that leads away from the state.
- A state becomes active as soon as it is entered. When a state is exited, it becomes inactive.
- As soon as a state is entered, the entry behavior (here: function 1) is executed. When a state is exited, the last thing to happen is the execution of the exit behavior (here: function 2).
- The state behavior of a state ("do" behavior) is the function (here: function 3) that is started directly after ending the entry behavior (here: function 1).

- A state can be exited through a transition only after the entry behavior (here: function 1) has been fully executed.
- The initiation of function 4 by a trigger under an optional guard condition does not lead to an external state change even if the behavior of a function (here: function 5) is part of the list of deferred behaviors of the state.

Finding states

If the theoretical viewpoint from Section 4.4.4.2 is followed literally, in general, an object can have many, sometimes even an infinite number of states. In order to reduce this number of states to a reasonable level, two procedures are recommended:

- Omit attributes that are irrelevant for the state observation.
- Form equivalence classes of possible attribute values.

Looking at the example from the introduction, for the task in question we can consider whether for the object *car,* the attribute *color* is relevant. If not, it does not have to be included in the consideration of the state.

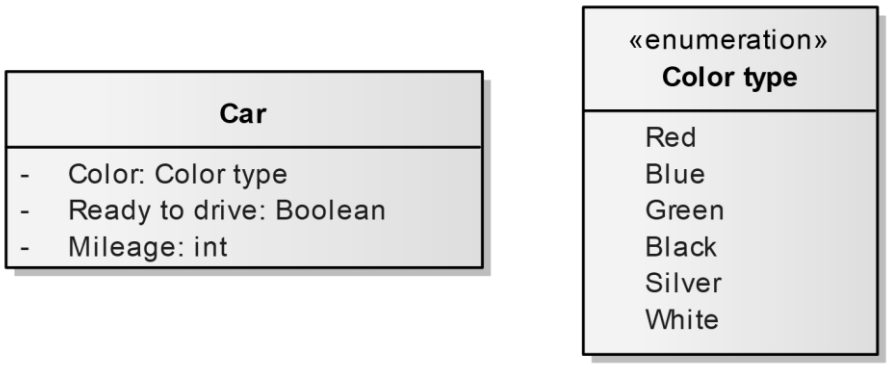


Figure 57: Definition of a car (c)

Equivalence classes are introduced to decide whether the possible values of the attributes can be divided into certain areas. The object under investigation will behave in the same way regardless of exactly which value is selected from a range of values of an attribute. Therefore, it may seem appropriate to divide the mileage of a car into three areas: "low", "medium", and "high". This reduces the number of theoretical states to a finite number.

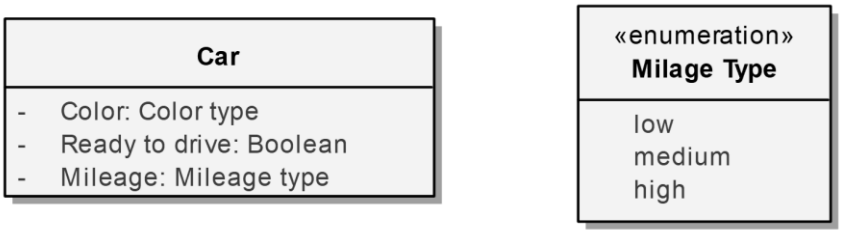


Figure 58: Definition of a car (d)

The number of the resulting states can be reduced further by grouping states into technically useful groups.

When considering systems, states are identified by the following rule: system states differ from each other by the fact that the system under development shows different behavior to the outside depending on which state it is in. These differences are reflected mostly in the fact that an actor will be able to use different features of the system based on the state it is in.

4.4.4.2 Transitions

Syntax and semantics

In UML, a transition is represented by an arrow with an appropriate name. It connects an initial state to a target state.

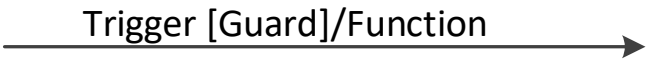


Figure 59: Notation of a transition

The naming of the edge consists of the following optional elements:

- **Trigger:** The trigger for the transition. The individual triggers are separated by commas.
- **Guard:** A condition that must be true before the transition is executed upon receipt of the trigger. The guard condition is listed in square brackets.
- **Function:** The function that is executed when passing through the transition.

Here, note that by definition, going through the transition must not consume any time. Therefore, only "short" functions should be referenced (such as the starting or stopping of an engine).

Normally, the output state is exited by going through a transition and then another state is reached as the target state. However, it may be the case that the source and target state are the same. This particular type of transition is referred to as a self-transition.

The transitions are triggered by a trigger and executed if the corresponding guard has a value "true". Of course, this only applies if a guard is specified in the transition.

UML acknowledges numerous types of triggers. In requirements modeling, it is mainly the following two types of trigger that occur:

- **Signal trigger:** A signal trigger is an incoming signal to the active state which triggers the execution of a transition. Therefore, the terms "trigger" and "signal" are very often used interchangeably.
- **Time trigger:** With a time trigger, you can trigger a transition at a certain time or after a certain period of time. OMG UML/SysML use the keyword AFTER, which is listed instead of the name of the transition.

In addition to being triggered by a trigger, a transition can be traversed without the trigger. This is the case as soon as the guard is "true" if you have listed only a guard and no trigger on the transition.

A guard can check the validity of certain values, such as **"x = 5"** or ranges of values **"x> = 10"**, as well as statements such as "x is located on the desktop" ("x" in this case can represent a parameter that results from an operation or a signal. It can also be a system variable). It is crucial that the guard represents a Boolean condition. The truth of this condition can be evaluated at any time, that is, the condition has either **"true"** or **"false"** as a value at any time.

The receipt of a signal and the consequent triggering of a transition are executed only when the object of observation is in a state which includes the signal as a trigger and the transition leads away from it. If no such transition is defined for the current state, then the signal is discarded. In the current state, this signal is defined as "to be delayed" (defer). The signal is reset and once the next signal arrives, it will be used again.

Transitions provide a transition from a source to a target state. If two transitions have the same initial state, they should be distinguished by different triggers or with the same trigger but different guards. This is not a prerequisite, but it makes the execution of the resulting state machine deterministic.

Finding transitions

There are two different approaches for finding the transitions:

- Identification of transitions from outgoing states
- Identification of transitions from incoming signals

The first approach is very intuitive because you have already given some thought to the identification of the states, why two states are to be differentiated, and when to switch from one state to another. An example of this approach is when you examine the use cases you have assigned to the states as functions. Is the postcondition formulated defined as a state? If so, the transition should lead to that state because the system should take on exactly this state (see also Section 4.2).

The second approach is more methodical. This is about whether and how the use case responds to an external signal when the system is in a particular state. This is repeated for all incoming signals and potentially for all states. This approach is more likely to be used in the consideration of a more technical system, in which perhaps the interfaces are specified with the external interfacing systems.

The second approach for finding transitions is also closely related to the modeling in the scenario view (see Chapter 5). A message that is received from the object under investigation will generally result in one or more state transitions during the processing of the message. Therefore, modeling of the scenario view is also used to locate and verify the state transitions in a state machine.

4.4.4.3 Initial state

Syntax and semantics

Figure 60: Notation of an initial state

Whenever a state machine is started, the first transition is the transition that leads from an initial state to a state. Because a system must always be in a certain defined state, the initial state is also referred to as a "pseudo" state. The system is never in such a state at any point in time. This means that no guard and no trigger may be listed on the output edge of an initial state.

In addition to an initial state in a state machine, initial states can also exist in the composite states. Section 4.4.4.5 looks at this subject matter in more detail.

Finding initial states

Each state machine should have exactly one initial state and finding it is not difficult. You simply draw the first state that the system is to take after the start.

4.4.4.4 Final State

Syntax and semantics

Figure 61: Notation of a final state

If the final state is reached, the execution of the overall state machine is terminated. After reaching the final state, no additional activities are executed. Therefore, there can be no outgoing pointer from final states. Technically, the final state can be seen as the end of the life cycle for the modeled object under investigation.

Finding final states

At this point, we have to consider and analyze in detail the specific features of the object under investigation. Which of the life cycles is relevant for meeting your requirements?

For example, if software is considered solely while it is being run, then exiting the software equates to the final state. However, if we are considering an embedded system over the entire period in which it is "built" into its environment, no final state is needed because the system may never terminate (see also the example in Section 4.4.3).

In addition, final states also exist in the composite states, which are presented in the next section.

4.4.4.5 Composite state

Composite states are composed of one or more states.

Syntax and semantics

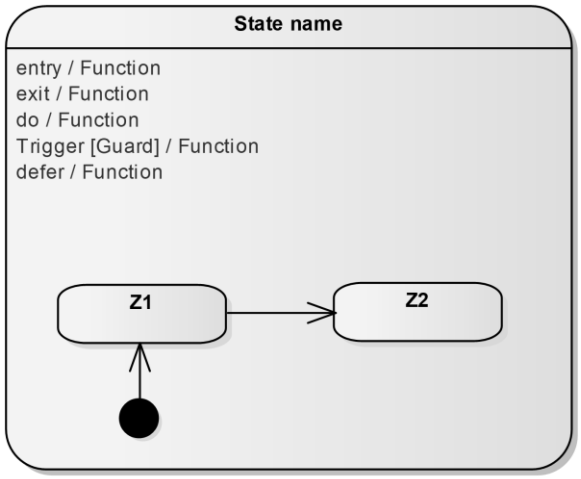


Figure 62: Notation of a composite state

The states included in a composite state are referred to below as substates. All types of states are possible as substates of a composite state. This means that in addition to the simple states and pseudo-states, you can also use a composite state. This allows you to define a hierarchy of states. The leaves in the resulting state tree are the simple states; the inner nodes are composite states.

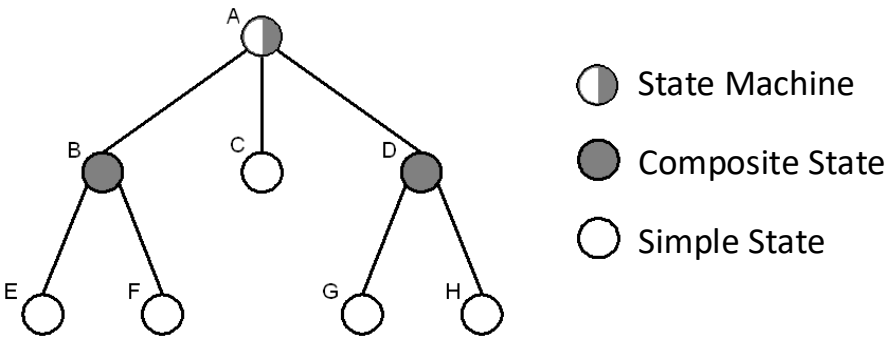


Figure 63: Hierarchical states

The root of the state tree is an exception because in a fully defined model, it always represents a state machine. It describes the behavioral description of the object under observation as it is seen from the outside.

As described in Section 4.4.2 above, one state must be active in a state machine. This rule must be met at all times. If the state is a composite state, one of its substates is active. Since this substate may in turn be a composite state, the definition of the active states continues downwards in the hierarchy until a simple state can be referred to as the active one.

Entering composite states

The possibilities for entering a composite state are described in the following figure.

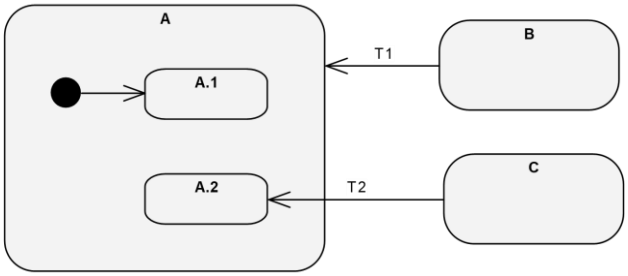


Figure 64: Entry into composite states

Semantics when entering composite states:

- **Default entry (trigger T1):** If state A is entered starting from state B, the start node is passed through and the active state is A.1.
- **Explicit entry (trigger T2):** If state A is entered starting from state C, the starting node is not passed through and the state A.2 is entered directly.

Modeling provides the history construct as another possibility for entering composite states.

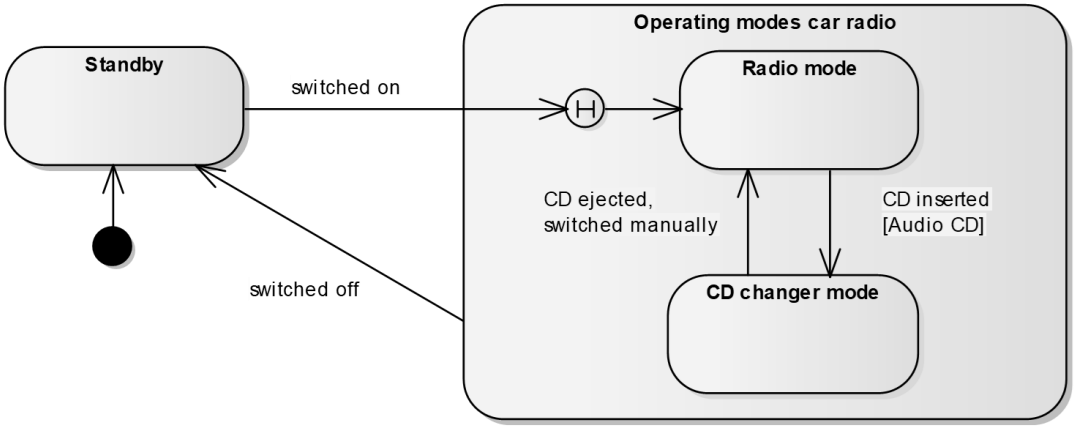


Figure 65: Shallow history

If the state "Operating modes car radio" is entered, the state which was active the last time this state was exited becomes active again. It is only in the special case of the first-time entry (i.e., no is history available) that the "Radio mode" is active. In the picture, the "Shallow history" is represented by H.

If there is a deeper hierarchy of composite states, the "Deep history" may be used. This not only remembers the substate of the upper level but also ensures that all nested substates (down to the leaf level) are remembered. This deep history is represented by H*.

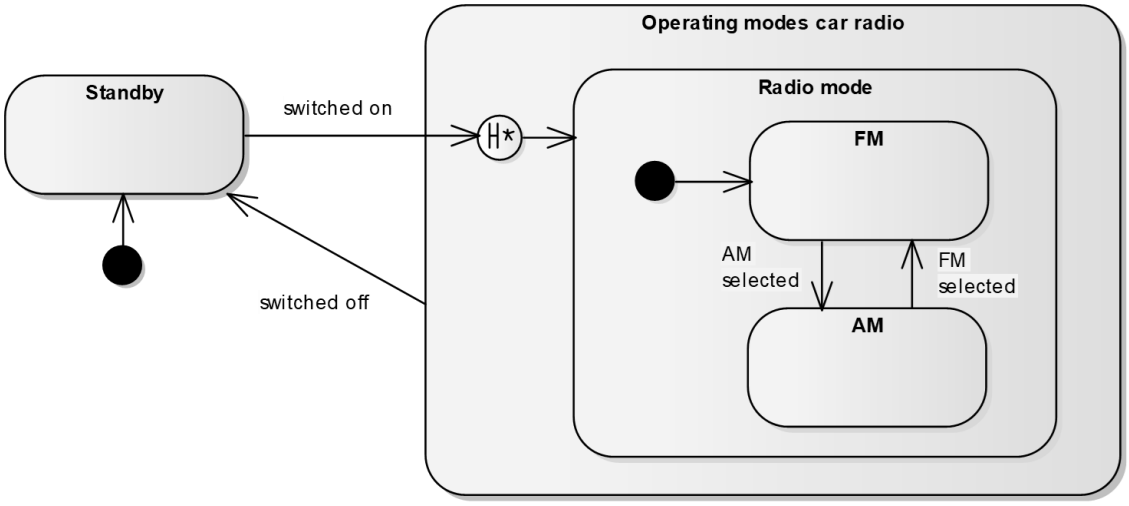


Figure 66: Deep history

Exiting composite states

There are also different ways to exit composite substates.

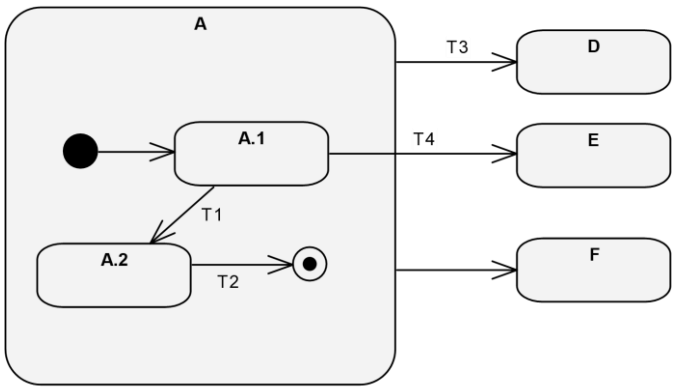


Figure 67: Exiting composite states

Exiting a composite state:

- **Reaching the final state (trigger T2):** There must be a transition from state A without a trigger which is executed. The next active state is F.
- **Transition of a substate (trigger T4):** This corresponds to the logical semantics: if A.1 is active and signal T4 is received, state E becomes active.

- **Transition of the composite state (trigger T3):** Regardless of which substate is active (A.1 or A.2), as soon as the trigger T3 occurs, state A is exited. The strength of this modeling construct is demonstrated here. A state hierarchy emphasizes abstraction as a technique for coping with complexity because the behavior on the upper level is defined completely independently of the situation within A.

Finding composite states

Using composite states becomes easy with the following rule: if the system should exhibit similar behavior (exiting the state, calling functions) in several different states, these states may be combined into a composite state. However, it is not permissible for one state to belong to several different composite states. In this case, you have to determine (based on application logic) how to resolve this conflict.

In general, however, composite states arise relatively naturally when we look at the modes of the application. For example, a fan has two states at the upper level: "on" and "off". The "on" state can then be subdivided further based on the chosen speed (slow, fast).

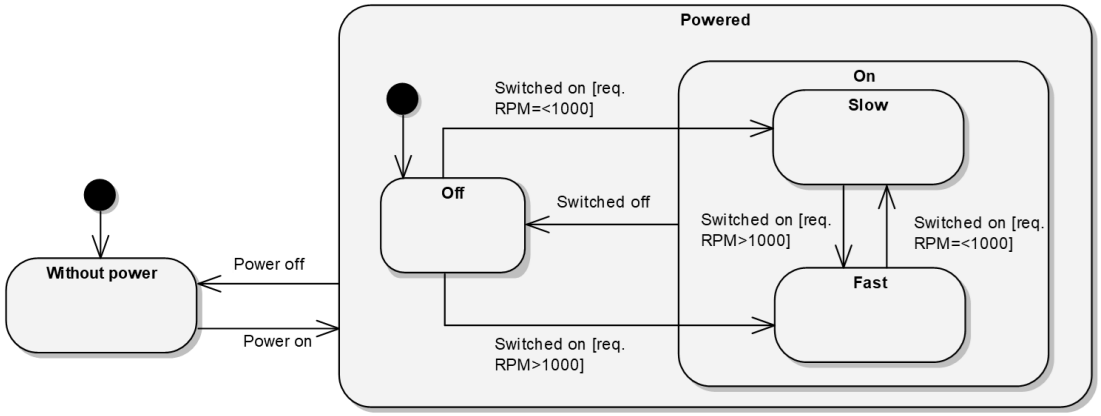


Figure 68: States of a fan

4.4.4.6 Substate machine

Syntax and semantics

A substate machine is represented as a simple state. However, there are two possible extensions to a simple state. The name of the substate machine and the name of the state it is associated with are separated by a colon. The other option is to put a shape that resembles a pair of glasses at the bottom right.

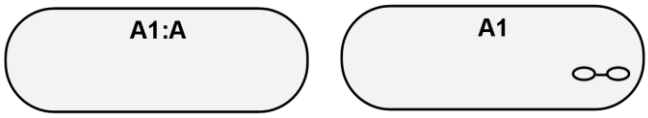


Figure 69: Syntax of a substate machine

With the introduction of the substate machine, the idea of hierarchical Statecharts, as introduced by composite states, is continued. The lower-level states of a composite state are shown graphically as a separate state machine (in a separate diagram). At a higher level, the state machine is referenced via this substate machine.

In order to also use the transition mechanism described in Section 4.4.4.5 in substate machines, entry and exit points are introduced. With these model elements, both an explicit entry and a transition can be modeled in a substate. This continues the concept of abstraction as described in Section 4.4.4.5.

Figure 70 shows the transformation from a composite state into a substate machine.

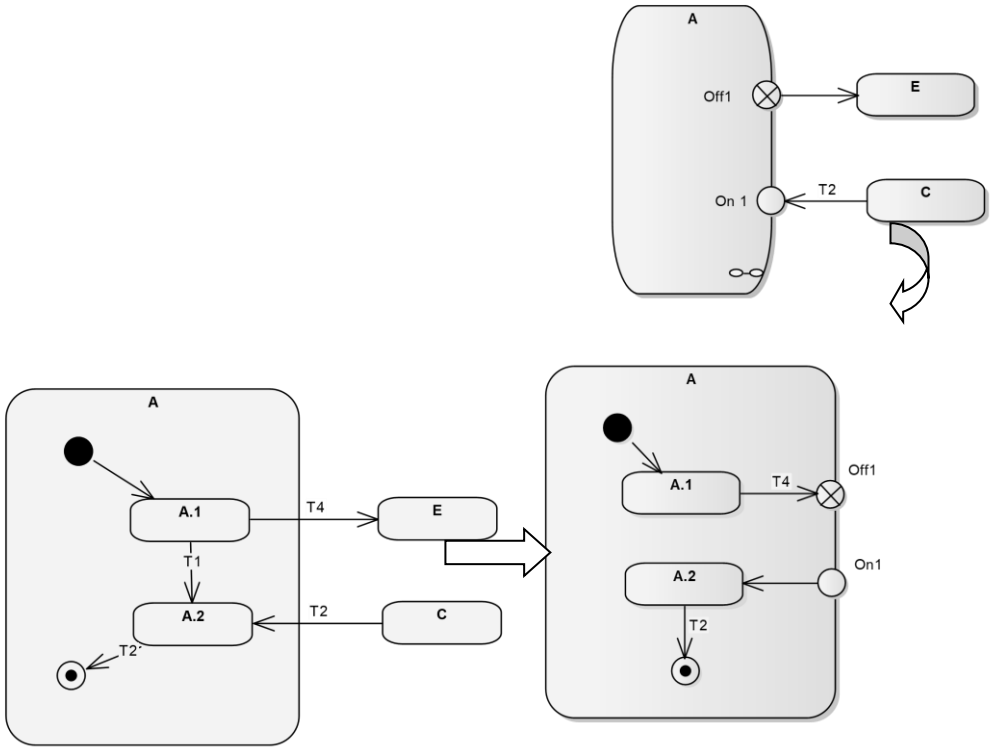


Figure 70: Use of entry and exit points

The left-hand part models a composite state; the right-hand part shows the use of a substate machine. Note where the triggers T4 and T2 are listed in the solution on the right. An example of the distribution of guards is given in the example section below.

Finding substate machines

For the identification of substate machines, the same heuristic can be applied as is used in identification of composite states (described in Section 4.4.4.2). In addition, note that multiple abstract state machines can be used in one substate machine. The diagrams can be made clearer using this concept.

Example:
As an example of this type of reduction of the complexity, the state machine of a fan is shown with an abstract state machine and a refinement of the state "On".

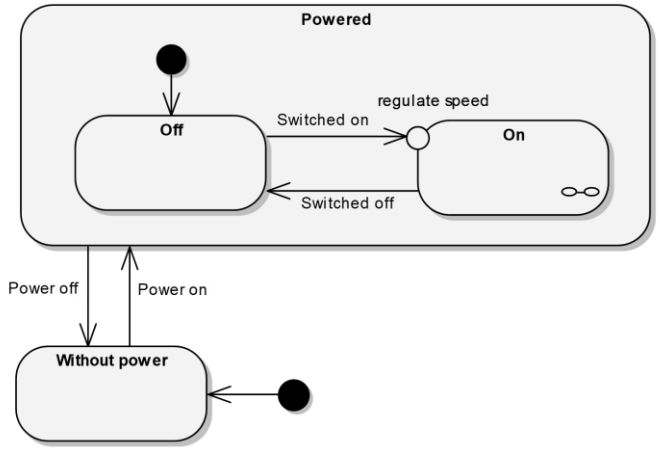


Figure 71: State machine of a fan

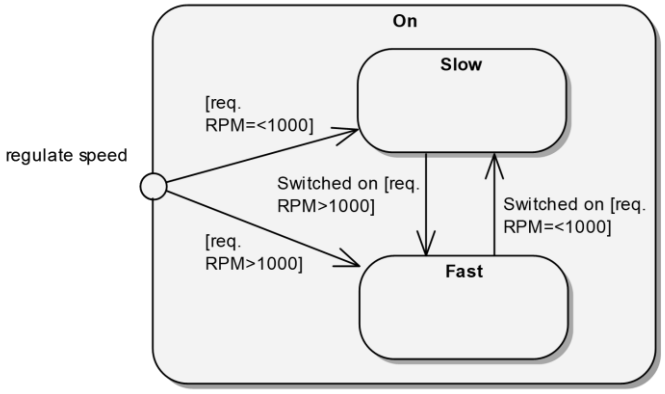


Figure 72: Hierarchical states of a fan

4.4.4.7 Orthogonal Regions

Using orthogonal regions, it is possible to define two or more parts of a state machine that can respond independently to events.

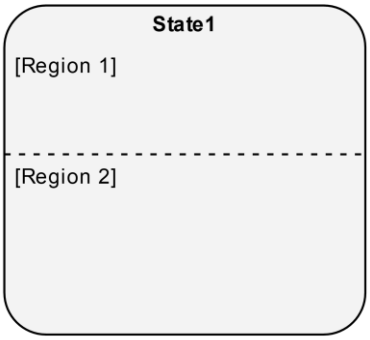


Figure 73: Syntax of orthogonal regions

A state can be divided into several orthogonal regions. Each region can have its own state machine, similar to the composite states model. This allows the opportunity to reduce the number of states if states can be distributed over several independent sets.

By way of explanation, let us look at the following example of an infotainment system which offers both a radio and a navigation system (see Figure 74). After turning the infotainment system on, the radio and the navigation can be switched to standby independently. Furthermore, the navigation can be set to destination entry or to the route guidance. Regardless of the navigation, the radio can be in radio mode or in CD changer mode.

These six possible states for the two parts would result in a total of nine substates (3 times 3, this will become clearer later on), presuming the system is in the *active* mode (after activation). Since each of the three states are independent of each other, the state *active* can be split into two orthogonal regions. The following state machine shows the result of this.

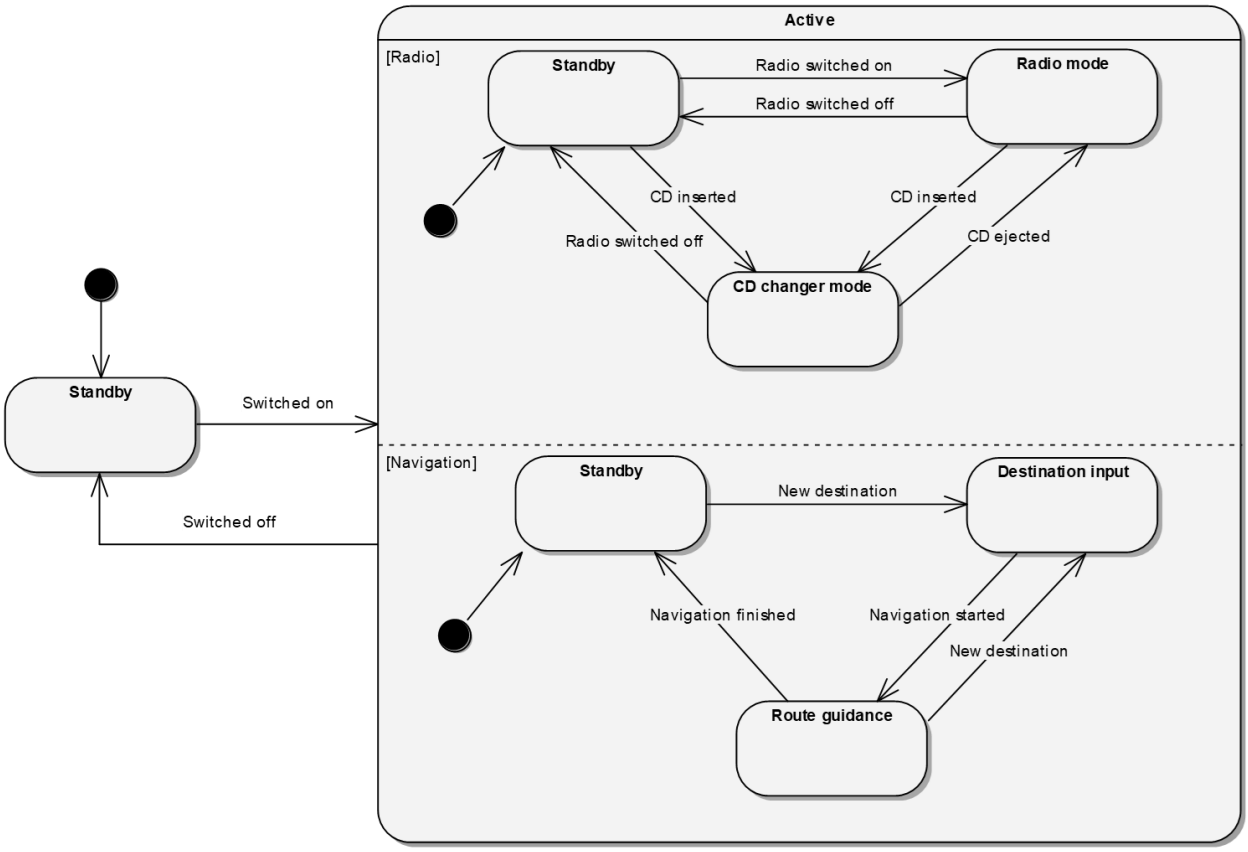


Figure 74: Orthogonal regions of an infotainment system

For the independence of the states, the following rules must apply:

- The behavior in a region is independent of the current state in the other region.
- Transitions across the boundaries of the regions are not allowed.

Note that even with the use of orthogonal regions, the paradigm mentioned in Section 4.4.2 is not violated. The system is still in exactly one state at any time but the state results from the combination of the active states in the individual regions.

The example given above uses one possibility for exiting the active state just as in the composite states. For entering the active state, the modeling construct of parallelization is used to express which two substates the system should adopt at the same time. In addition to these options, there is a variety of other entry and exit options. For a complete overview, see [BoRJ2005].

The following figure shows the state machine of Figure 74. We can clearly see that from the six states modeled, nine states are now being derived. The number of transitions increases even further.

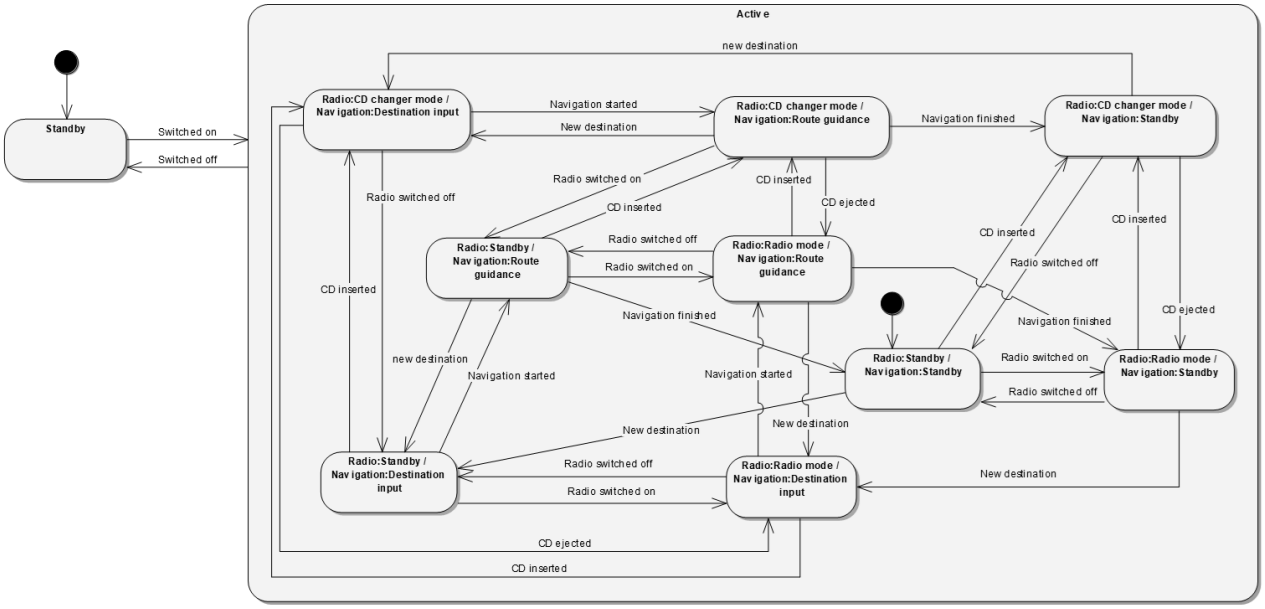


Figure 75: Resolved orthogonal regions

Finding orthogonal regions

Finding regions and recognizing that orthogonal regions can be formed is not always easy. It is good practice to start modeling without orthogonal regions. When the state machine becomes too complex, check whether perhaps the names of the states indicate certain orthogonal regions. In many cases, an indication for those regions is that parts of a state (refinement) are discussed in several independent parts of the state machine.

4.4.5 Typical state machines/modeling scenarios

4.4.5.1 Generic state machines for technical systems

The following figure shows a generic solution which can serve as a template for state machines of technical systems.

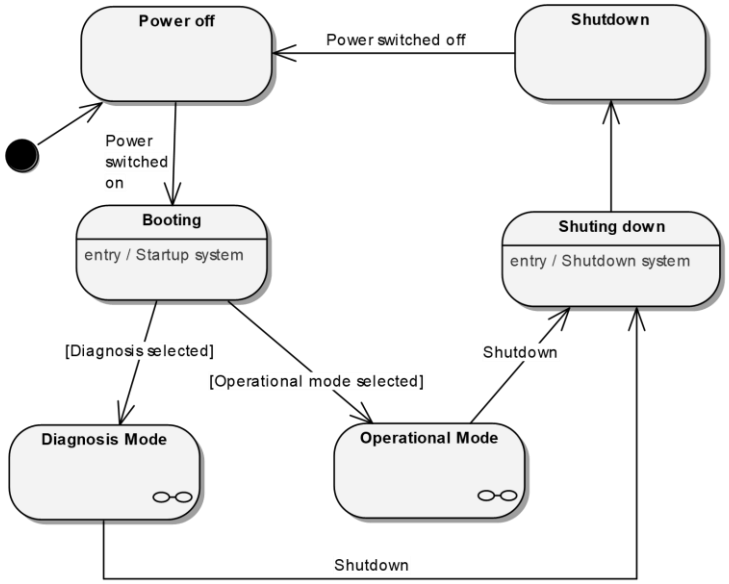


Figure 76: Generic state Machine of a technical system

In this machine, the two states "Diagnosis Mode" and "Operational Mode" should be further refined. However, these refinements are highly dependent on the system under development, so no further statements about the form of these states can be taken at this point.

Even more states can be integrated in this state machine if required. In infotainment systems in the automotive industry, for instance, a "Driving Mode" can be defined in which the system does not accept inputs. This state would be parallel to the "Diagnosis Mode" and "Operational Mode".

4.4.5.2 States of Objects of a Business-Oriented System

As a typical example of the states of an object in a business-oriented system, we will use the object *request for leave*. Figure 77 shows the state machine of this object, whereby the full definition of triggers, guards, and functions is omitted:

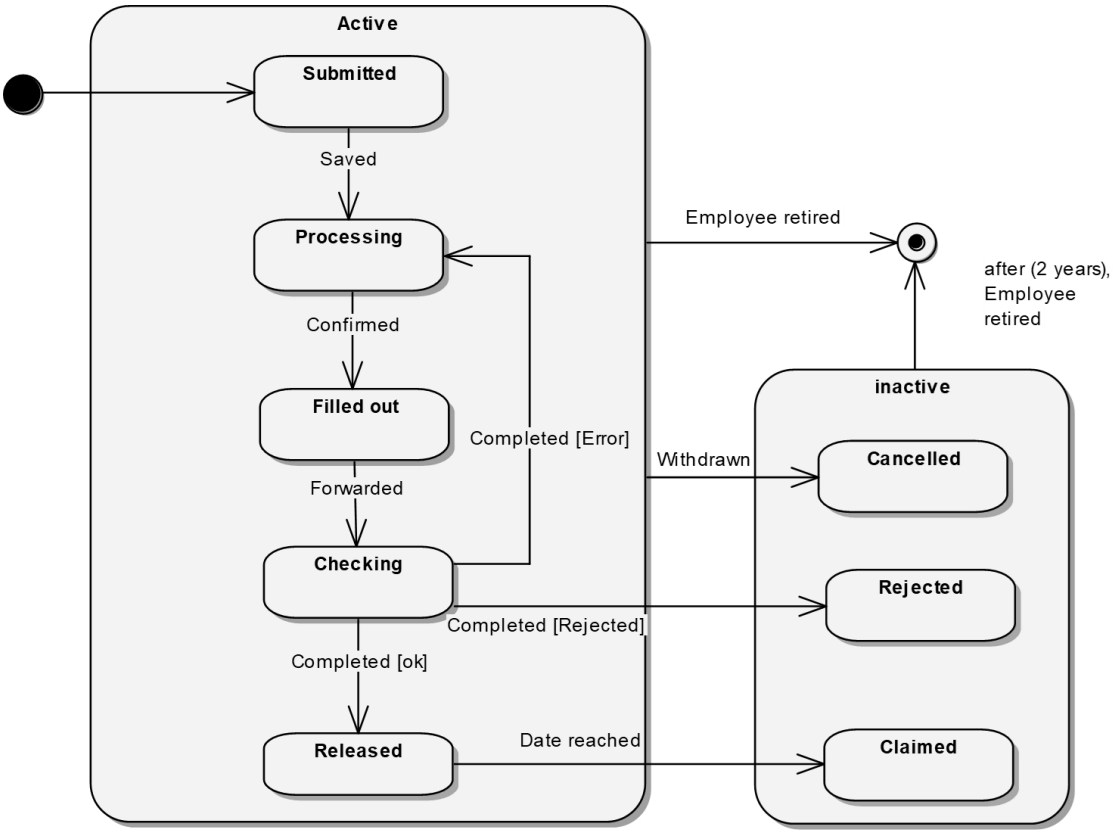


Figure 77: States of a request for leave

As we can see in the machine, the states of the object correspond to time periods in which the request for leave is stable (for some time). This also corresponds to the possible stages of processing by use cases because a use case contains a complete interaction between an actor and the system. As a result, the states of the request for leave must be stable after a use case is completed. From a technical perspective, this means that this information must be stored in the database so that the logical implementation knows which steps are allowed for a specific request for leave.

The close relationship between the states and the use cases for processing such a request for leave can be expressed in another way: the states of the object specify the postconditions that have been defined in a use case.

4.5 Further reading

Data flow perspective

- DeMarco, Tom: Structured Analysis and System Specification, Yourdon Press, Prentice Hall, 1979

Control flow perspective—in particular, activity diagrams

- Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, 2004
- Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley 2005.

Use case modeling and specification

- Jacobson, I.; Christerson, M.; Jonsson, P.; Oevergaard, G.: Object Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Reading, 1992.
- Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, 2004.
- Cockburn, Alistair: Writing Effective Use Cases, Addision Wesley, 2000.

State perspective

- Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, 2004

5 Scenario modeling

Today, scenarios are an essential tool in requirements engineering, for example, to specify the system vision and goals of stakeholders or to describe the added value created for the users of the system. Scenarios have the character of examples which look at the use of the system under development by humans or other systems (see, e.g., [Caro1995]).

Besides their use for the exemplary description of the use of the system under development, scenarios can also be used to specify functional requirements precisely. In this case, in the associated scenario view, all the scenarios that occur in the system usage are specified at a high level of precision—for example, through UML sequence diagrams or Message Sequence Charts according to the ITU standard [ITU2004].

5.1 Purpose

Since the early 1990s, scenarios have been used in requirements engineering to support the systematic specification of requirements (see, e.g., [Pott1995]). If the starting point for requirements engineering is the raw system vision or the goals of the stakeholders, in many cases it is difficult to immediately specify the requirements of the system completely and correctly based on that vision or those goals (see, e.g., [DaLF1993]).

This key insight led to the use of scenarios in requirements engineering. Scenarios focus on the interaction-based view which is a specific behavioral view of the functional requirements of the system. In this view, the behavior of the system is described by sequences of interactions between communication partners. In the center of the interaction-based view are the communication partners that represent either systems or individuals in the system context or the system under development, and the interactions between these communication partners.

An **interaction** between communication partners is a sequence of messages exchanged between these partners. These messages can be information or data that is exchanged via communication channels between the communicating actors. Moreover, requirements engineering also considers messages in the form of tangible flows between communication partners in interactions, for example, a material flow or cash flow between communication partners.

A **scenario** is an interaction between communication partners (often between the system under development and actors in the system context) that leads to a desired (or possibly undesired) result. Scenario modeling is often used to specify the system vision and goals of stakeholders with regard to the desired use of the system. Scenario modeling is not normally limited to only the interface of the system under development in the form of the direct message exchange between actors and the system but also considers messages that are exchanged between actors in the system context. Thus, scenario modeling is not only the modeling of the requirements of the system under development, but also the interaction context of messages which are exchanged between actors and the system under development.

In requirements engineering, the added value to an actor in the system context is often seen as an essential result of a scenario. The following example illustrates a simple scenario described in natural language which documents an interaction between a person (John) and an online store so that John can make a purchase.

Example Scenario "Shopping in an online shop":
In the product catalog of the online shop, John chooses the desired products and then confirms that he would like to finalize the purchase. The online shop shows John the selected products including the quantity and price and the total of the purchase. The online shop asks John to confirm the purchase.
After John has confirmed the purchase, the online shop asks for the shipping address. John enters the desired shipping address and confirms it. After confirmation of the shipping address, the online shop asks John for the payment information. John enters the payment details and confirms them.
The online shop then displays the complete order including shipping address and payment details and asks John to confirm this order. John confirms the order, whereupon the online shop displays an order confirmation.

The associated added value that the actor (John) gets through the use of the online shop is that John can order the desired products via the Internet.

5.2 Relationship between scenarios and use cases

There are various types of scenarios in requirements engineering. An extensive analysis of the different types of scenarios can be found in [RAC1998]. The following paragraph presents two frequently found differentiations of scenarios and the related types.

One common differentiation of scenarios distinguishes between main scenarios, alternative scenarios, and exception scenarios. This distinction is a key element of use case-based approaches (such as [JCJO1992]), in which scenarios that relate to a specific added value are grouped within a use case and are documented complementary to each other (see Section 4.2.5). The use of main, alternative, and exception scenarios is not necessarily limited to use case-based approaches.

- A **main scenario** is a scenario that describes a predominantly occurring sequence of interactions to achieve a specific result (e.g., a specific added value).
- An **alternative scenario** is a scenario that describes an alternative sequence of interactions to achieve the specific result in relation to a main scenario.
- An **exception scenario** is a scenario that describes a sequence of interactions that must be executed if a defined exception event occurs. In requirements engineering, exception scenarios are specified to handle exceptional situations in operations in a controlled manner, often in addition to main and alternative scenarios.

In practice, the number of exception scenarios is in most cases considerably larger than the number of alternative scenarios of a main scenario. This is because the exception scenarios (and associated exception events) should preferably cover all situations that can occur

during the execution of the main or alternative scenarios and that prevent a further successful execution of the corresponding scenarios (or the associated use case, see Section 4.2) in the operation of the system. Each exception scenario specifies a controlled exception handling in response to a defined exception event.

5.3 Approaches to scenario modeling

The modeling of scenarios allows the documentation of extensive and complex situations which involve the interaction-based behavior of the system in an easily understandable and structured way. Diagram types that allow the documentation of an arrangement of interactions between communication partners in visual form are particularly suitable for modeling scenarios. Today, sequence diagrams are often used for modeling scenarios. In sequence diagrams, the communication partners involved in the interaction sequence are arranged in the horizontal dimension.

The interactions between the communication partners are modeled in the order of appearance in the vertical dimension. In this way, scenarios from use cases can also be specified in more detail through diagrams (see Section 4.2).

In the telecommunications industry, Message Sequence Charts (MSCs) of the International Telecommunication Union (ITU) according to the standard ITU-T Z.120 [ITU2004] are used. The high degree of formalization of MSCs offers far-reaching possibilities for automatic processing such as quality inspection (e.g., to check freedom from contradictions and completeness) or generative approaches for development.

The use of h (high-level) MSCS (similar to the interaction overview diagrams in UML 2) allows appropriate structuring of extensive and complex models in the scenario view. The ITU-T Z.120 standard came into force in 1992 and has been subject to continuous improvement ever since. In particular, it has heavily influenced the **sequence diagrams of UML** [OMG2010c, OMG2010b] and the **sequence diagrams of SysML** [OMG2010a].

The use of UML/SysML sequence diagrams has the advantage that UML and SysML are much more widespread in practice than competing modeling approaches, such as those of the ITU. Moreover, through the metamodel of UML/SysML, scenarios modeled in UML/SysML sequence diagrams can be integrated with other views of requirements modeling if UML or SysML diagram types are also used in these views.

Besides UML and SysML sequence diagrams, UML provides another diagram type, **communication diagrams,** which also allows scenario modeling. Compared to sequence diagrams, which focus primarily on the sequence of interactions between communication partners, UML communication diagrams focus on the visualization of the bilateral interactions between communication partners. The sequence of interactions is then indicated by sequence numbers added to the interactions.

5.4 Simple examples of a modeled scenario

Figure 78 shows the modeling of a simple scenario in the form of a UML sequence diagram (a) and a UML communication diagram (b).

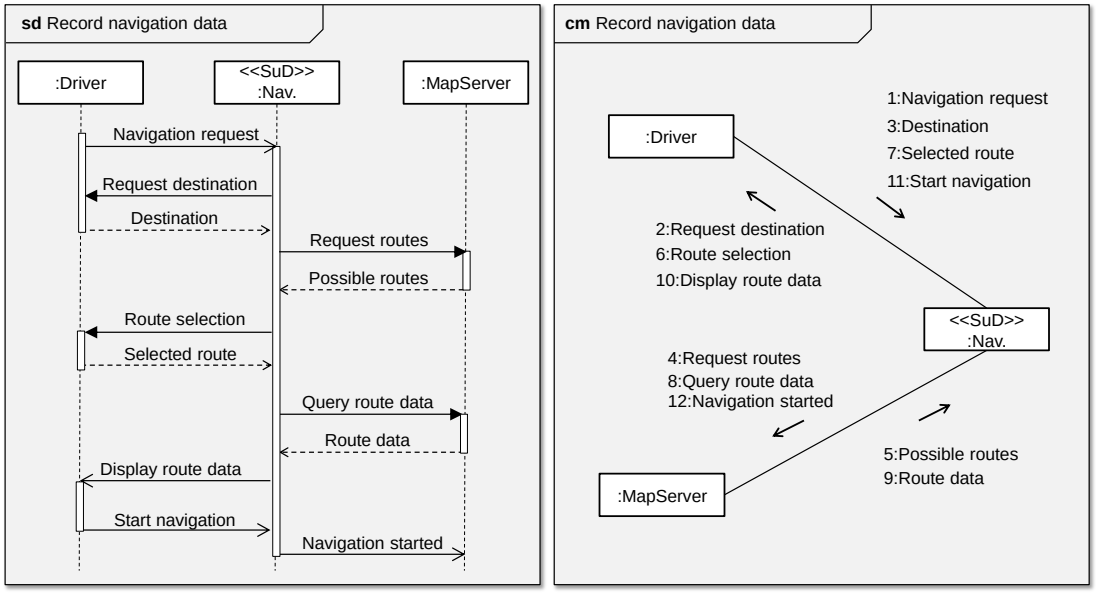


Figure 78: Modeling of a scenario with (a) sequence diagram and (b) communication diagram

Both diagrams model the scenario "Record navigation data". The name of the scenario is specified in the upper part of the frame. The keywords "sd" and "cm" respectively designate the diagram type used to model the corresponding scenario. In Figure 78, "sd" stands for sequence diagram and "cm" for communication diagram.

The sequence diagram on the left of Figure 78 shows a sequence of interactions between instances of the communication partners ":Driver", ":Nav" and ":MapServer" that must be executed so that the driver can enter the navigation data in the navigation device. The system under development is labeled with the stereotype <<SuD>> (system under development) to make the separation between the system and the actors in the system context clear.

As shown, in sequence diagrams the sequence of interactions is modeled in the vertical dimension. In the horizontal dimension, the instances of the communication partners involved in the given scenario are listed. The ":" in front of the name of the communication partner indicates that it is a concrete instance. The arrowhead indicates the direction of the message exchange.

The communication diagram on the right of Figure 78 also represents the scenario "Record navigation data". In this diagram, however, the sequence of the interactions is not documented in the vertical dimension but is instead annotated by specifying sequence numbers for the interactions.

With a line between communication partners, the communication diagram visualizes the existence of a direct communication relationship. The interactions occurring due to this communication relationship are documented by messages. Each of these messages is specified by a name, the associated sequence number of the message in the scenario, and the direction of the message flow.

In the visualization, communication diagrams place special emphasis on the communication relationship between two communication partners. In contrast, the temporal or logical sequence of interactions of scenarios is better visualized by sequence diagrams.

Due to the different priorities of the visualization, the requirements engineer must decide, depending on the situation, which one of the two diagram types is most appropriate for the respective use (↑ pragmatic quality).

If different uses are required, a scenario can be modeled in both diagram types. The sequence diagram or the communication diagram could also be constructed automatically from the diagram of the other diagram type. However, what is significant is that complex interactions (e.g., the conditional repetition of messages or alternative messages) cannot be represented by communication diagrams or only with a great deal of difficulty.

In the next section, the different model elements for scenario modeling with UML/SysML sequence diagrams or UML communication diagrams are presented, including an explanation of their specific relevance for modeling requirements. Further information about the model elements of sequence diagrams and communication diagrams can be found in [OMG2010b] or [OMG2010a].

5.5 Scenario modeling using sequence diagrams

Figure 79 shows the model elements of UML/SysML from OMG for sequence diagrams which are used for modeling scenarios.

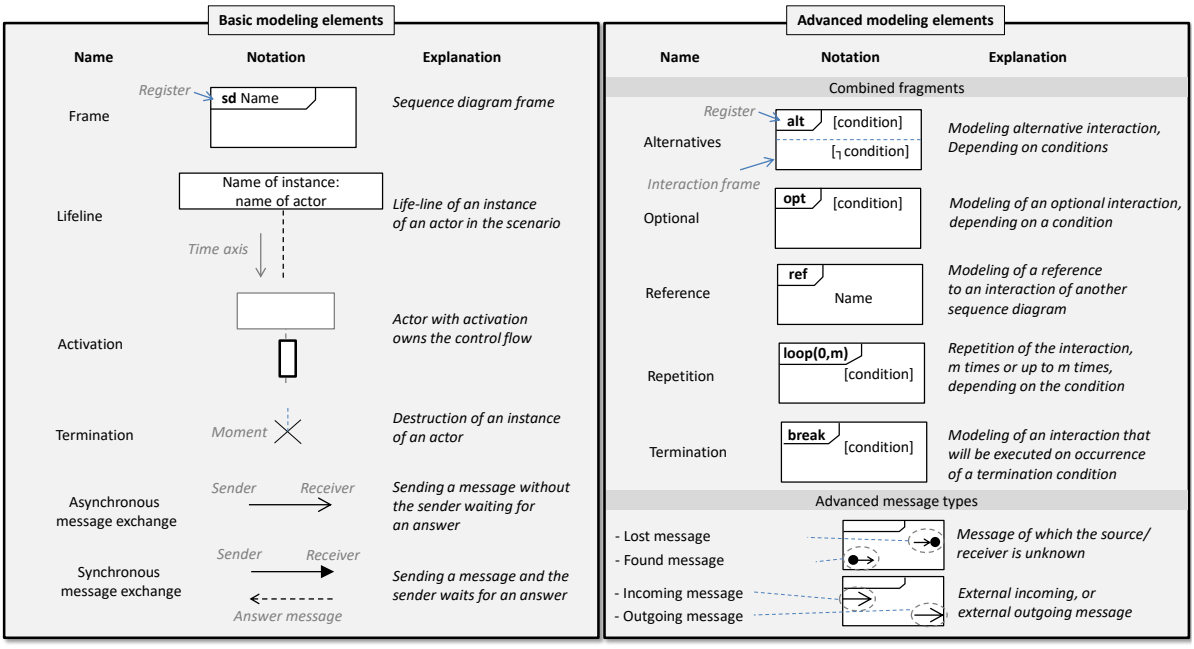


Figure 79: Model elements for scenario modeling using sequence diagrams

The left-hand panel of the figure presents the basic model elements, that is, those model elements that are essential for modeling scenarios with sequence diagrams. The right-hand panel of the figure shows the model elements that are used to model more extensive and more complex interaction relationships between communication partners.

5.5.1 Basic model elements

5.5.1.1 Modeling the identifiability and referenceability of a scenario

Sequence diagrams have an outer **frame** (interaction frame) which has the name of the scenario that is modeled by the diagram in a register in the upper left area.

The **name of the scenario** has the prefix "sd", which, as already explained above, indicates that the scenario is modeled by a sequence diagram. The use of frames means that the scenario can be identified and referenced by name, which in particular supports the management of different diagrams.

5.5.1.2 Modeling the communication partners of a scenario

A **lifeline** represents one instance of an actor within the scenario. The naming of the lifeline follows the pattern instance name**:** type name (e.g., Peter: Driver). When modeling scenarios, instance names are often omitted. However, instance names should be specified if it improves the understandability of the modeled scenario.

If several instances of a certain communication partner are needed in one scenario, each instance should be given a different instance name. This differentiation makes it clear that two different instances of an actor of a scenario are involved and that there is a direct message exchange.

The **activation** of a lifeline indicates that the respective communication partner has the control in the visualized period within the scenario, that is, the communication partner determines the control flow of the scenario. A **termination** in the lifeline of an instance signifies the destruction of the corresponding instance of the actor. Figure 80 shows an example of modeling a lifeline with activation and termination.

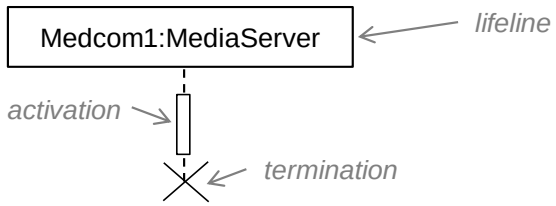


Figure 80: Modeling of lifelines and termination

5.5.1.3 Relationship of actors in scenarios to context models and use case models

The actors in the scenarios are also visible in use case diagrams and the context diagrams of the system, which means that the modeled scenarios can be integrated with the use case diagrams of the use case view (cf. Section 4.2) and the context diagrams (cf. Section 2.2) via the communication partners in the scenarios. Typically, the context diagrams are created before the scenario modeling, which means that the actors and interfaces documented in the context diagram can structure and guide the systematic creation of scenarios. Actors that occur in the scenario modeling but cannot be found in the corresponding use case and context diagrams indicate that the context and use case models are incomplete (cf. Section 4.2.3).

5.5.1.4 Modeling the message exchange within a scenario

The message exchange between two instances of communication partners within a scenario is visualized by an arrow. The direction of the arrow indicates the direction of the message exchange. There are two types of message exchange.

In an **asynchronous message exchange** between instances within the scenario, the transmitter sends the message to the receiver and does not wait for a corresponding response in the form of a message from the receiver. In scenario modeling, asynchronous messages are used, for example, when an instance wants to send information to one or more instances within the scenario and does not expect a response from the receiver.

In a **synchronous exchange of messages** between instances within a scenario, the sender of the synchronous message waits for a response message from the receiver. One use of synchronous messages in scenario modeling is when an instance within the scenario requests information from another instance. An example of this would be the synchronous message "Request personal identification number (PIN)" sent by the instance of an ATM to the instance of a user. The ATM then waits for the user to enter the PIN, that is, to send a response message with the PIN.

In scenario modeling in requirements engineering, the "message exchange" refers not only to data that is transmitted through a communication infrastructure between communication partners; a "message exchange" within a scenario may also represent the exchange of tangible or intangible entities—for example, the insertion of a credit card (tangible entity) into the ATM by the user.

Figure 81 shows an example for the modeling of both asynchronous and synchronous messages.

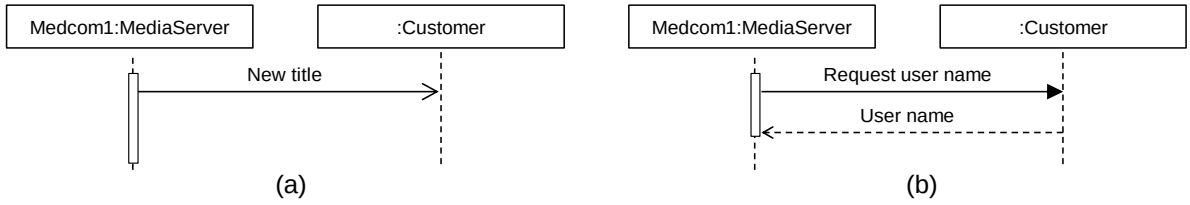


Figure 81: Modeling a) asynchronous and b) synchronous messages

Through message exchange, the sending communication partner can request a service from another communication partner. Again, the **service call** can be asynchronous or synchronous. With an asynchronous invocation of a service, the service is merely triggered by a message, that is, the calling communication partner does not wait for an answer. With a synchronous call, the transmitter waits for the corresponding response from the receiver once he has requested the service from another communication partner through a message.

A service call can also include its signature, which means that **input parameters** (arguments) and **return parameters** can be specified. Parameters are typically defined in the information structure view, which creates a relationship (integration) between the scenario view and the information structure view. Figure 81 also shows the use of the optional model element to represent the activation of a communication partner.

Figure 82 shows an example of the modeling of a service call with incomplete and complete parameters.

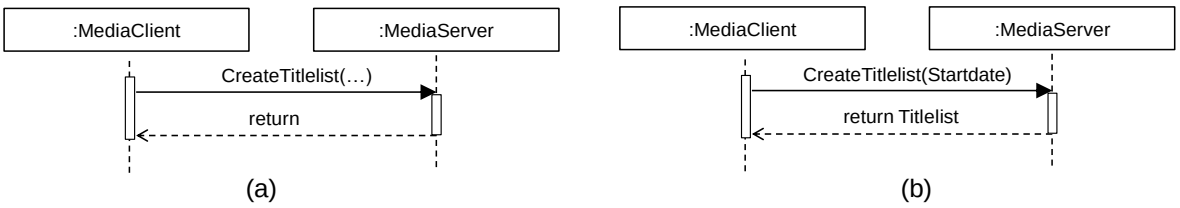


Figure 82: Modeling of a service call a) with incomplete and b) complete parameters

5.5.1.5 Relationship of messages in scenarios to state-oriented modeling, data flow-oriented modeling, and information structure modeling

The exchange of messages within a scenario represents the essential integration point to the diagrams of other views of the requirements of the system under development (cf. Figure 83).

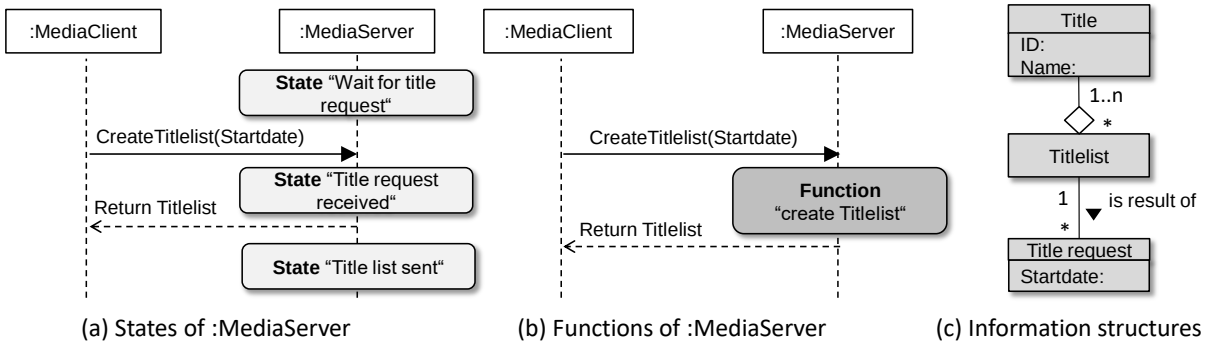


Figure 83: Messages in scenarios as an integration point with other requirement views

Relationship of messages to states in the state-oriented view

As shown in Figure 83 (a), both receiving and sending a message corresponds to a change in the state of the actor. In Figure 83 (a), for example, receiving the message "CreateTitlelist(Startdate)" corresponds with the state change of the communication partner ":MediaServer" from the state "Wait for title request" to the state "Title request received". Sending the message "return Titlelist" also results in a state change for ":MediaSever" (into the state "Title list sent").

At the same time, receiving this message results in a state change of ":MediaClient". The states of the various communication partners of a scenario and the state transitions can be modeled through diagrams of the state-oriented view, for example, through a UML state diagram (see also Section 4.4).

Relationship of messages to functions/activities in the data flow-oriented or control flow-oriented view

As shown in Figure 83 (b), there is a functional relationship between receiving a message and subsequently sending a message based on the system under development. The reason for this relationship is that the system has to execute a function based on the incoming message and, where applicable, based on locally available information in order to create the result message.

These functions (processes, activities) are typically modeled in the data flow-oriented or control flow-oriented view: the data dependencies and control flow dependencies between these system functions are modeled, for example, in one or more data flow diagrams and activity diagrams (see also Section 4.3).

Relationship of messages to classes, attributes, and associations in the information structure view

As shown in Figure 83 (c), the messages and any corresponding parameters are defined in the information structure view of the requirements. The corresponding information is specified, for example, in a class diagram which defines the information structure of the messages in detail, including the technical relationships to other messages that are

exchanged between the system under development and the actors in the system context (see also Section 3).

5.5.2 Advanced model elements

The use of combined fragments allows us to model large and complex interaction-based behavior in scenarios in an easily understandable way through sequence diagrams. UML or SysML distinguish between a number of different types of combined fragments.

Below, five types of combined fragments are presented which are very suitable for modeling large and complex interaction-based behavior in scenarios. Combined fragments are modeled through interaction frames within a sequence diagram. The type of the combined fragment and thus the corresponding meaning of the interaction within the combined fragment in relation to the surrounding scenario are specified via a keyword in the register of the combined fragment. In the vertical dimension of the sequence diagram (timing), the interaction frame is typically extended as far as the specific interaction takes place over time. In the horizontal dimension, the interaction frames of the combined fragments are extended as far as to include all instances that exchange messages within the specific interaction in the combined fragment.

5.5.2.1 Modeling alternative interactions of a scenario ("alt")

Alternative fragments are used to model alternative interaction sequences (i.e., an alternative behavior) of a scenario. Within the sequence diagram, a corresponding interaction frame is modeled with the keyword "alt" in its register.

The interaction frame is divided into two or more sections. For each of these sections, an explicit Boolean condition must be specified that determines when ("when" in the sense of a logical condition) the interaction in the corresponding section is executed. For one section, the condition "else" can be given, thereby specifying that the corresponding interaction is executed if none of the other conditions at the time of the potential entry into the combined fragment are true.

If this section is omitted, no interaction is executed if none of the conditions are true when the combined fragment is entered. The Boolean condition of each section is typically modeled over the lifeline of the instance within the scenario that has access to the value used to evaluate the Boolean condition. The Boolean condition can be arbitrarily arranged over the lifelines if the values are global values.

In formulating the conditions for individual sections of the alternative interaction of the scenario, it is important to make sure that they do not overlap from a logical point of view, that is, no more than one condition is true when the combined fragment is entered. If this is not the case, the associated scenario would have non-deterministic behavior (cf. Section 4.4). Figure 84 shows an example for the modeling of a combined fragment of the type "alternative".

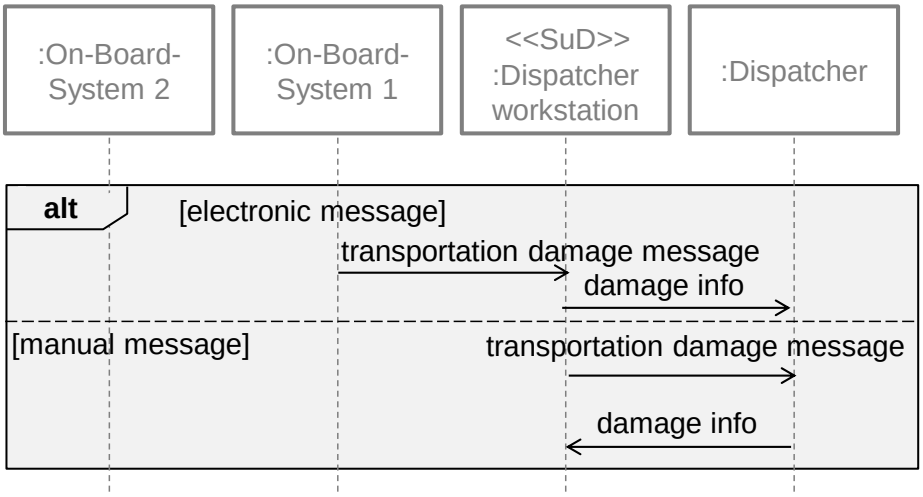


Figure 84: Modeling of a combined fragment of the type "alternative"

5.5.2.2 Modeling optional interactions of a scenario ("opt")

Optional fragments are used to model optional interactions (i.e., optional behavior) of scenarios. Within the sequence diagram, a corresponding interaction frame is modeled with the keyword "opt" in its register. In the interaction frame, an explicit Boolean condition should be specified that defines which condition must be true during the execution of the scenario at the time of the potential entry into the combined fragment. The interaction modeled in the optional fragment is then executed.

The Boolean condition is typically modeled over the lifeline of the instance within the scenario which determines whether the corresponding condition is satisfied or not. If the condition is not true at the time of the potential entry into the combined fragment, the corresponding interaction (or the associated exchange of messages) does not take place during the execution of the scenario. An optional combined fragment may be regarded as an alternative combined fragment that has only one section with a corresponding condition. Figure 85 shows an example of the modeling of a combined fragment of the type "optional".

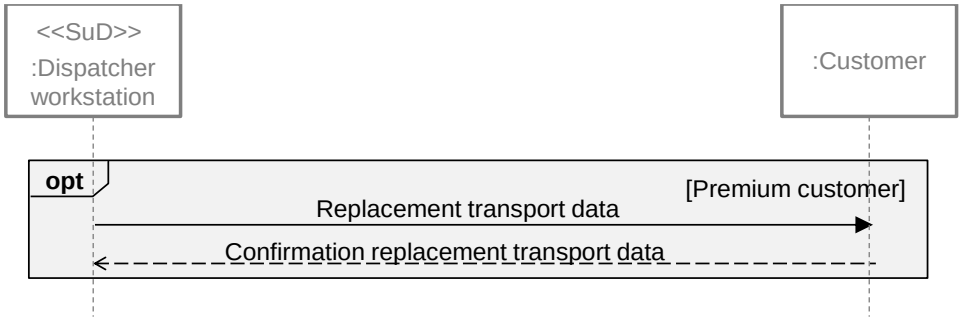


Figure 85: Modeling of a combined fragment of the type "optional"

5.5.2.3 Modeling abstractions of interaction sequences of a scenario ("ref")

Sequence diagrams provide the ability to abstract from combined interaction sequences of a scenario by referring, at the appropriate position in the sequence diagram, to another sequence diagram which models the corresponding interaction of the scenario. For this purpose, a combined fragment is modeled in the sequence diagram at the position at which the abstracted interaction occurs. The combined fragment is then characterized in its register with the keyword "ref".

The name of a scenario is specified in the center of the fragment. This is the scenario which contains the detailed interaction which, during the execution of the parent scenario, is integrated into the interaction of the scenario at the position indicated by the combined fragment. The use of combined fragments of this type is particularly appropriate when large or complex interaction behavior of a scenario has to be modeled.

This allows the requirements engineer to extract technically connected interactions of a complex scenario into a separate sequence diagram. The use of combined fragments of the type "reference" is also appropriate if certain interactions (such as the interactions to authenticate a user on the system) occur in an identical manner in several scenarios.

When modeling interaction sequences in separate sequence diagrams which are referred to in other sequence diagrams by a combined fragment of the type "reference", the requirements engineer must ensure that the partial scenario that will be included is compatible with the parent scenario. For example, no instances that do not occur in the parent scenario or in the corresponding sequence diagram may occur in the partial scenario. Figure 86 shows an example of the modeling of a combined fragment of the type "reference".

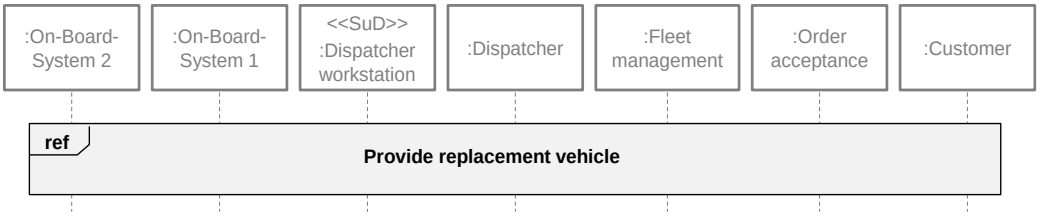


Figure 86: Modeling of a combined fragment of the type "reference"

5.5.2.4 Modeling repetitions of interactions within a scenario ("loop")

To express repetitions of interactions within a scenario, a corresponding interaction frame is modeled within the sequence diagram with the keyword "loop" in its register. In combined fragments of this type, the number of repetitions is specified either by `loop ([number])` or by `loop ([min, max])` with a lower (min) and an upper (max) limit on the number of repetitions.

In the latter case, the limits for the repetition specify that the interaction is repeated within the interaction frame at least min and at most max times. In this case, the repetition of the interaction within the interaction frame is also specified by a Boolean condition.

If the interaction within the interaction frame of the scenario is repeated min times, the repetition is discontinued if the evaluation of the Boolean condition is false when re-entering the interaction frame of the combined fragment.

If the Boolean condition is true for each entry into the interaction frame, the repetition of the interaction is completed after max runs. Figure 87 shows an example of the modeling of a combined fragment of the type "loop".

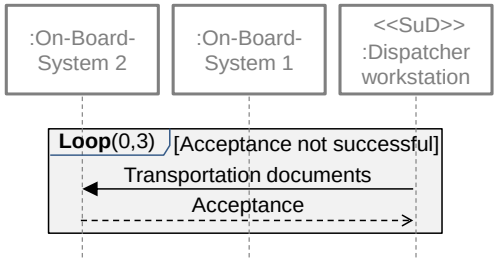


Figure 87: Modeling of a combined fragment of the type "loop"

5.5.2.5 Modeling the termination of a scenario ("break")

During the course of a scenario, situations may arise that prevent the successful execution of the scenario. To represent the necessary exception handling from a technical point of view in such cases, the interaction for the exception handling can also be modeled in sequence diagrams. The termination fragment contains an optional Boolean condition and an optional interaction that is executed to handle the termination if the condition for the termination is true.

If no explicit termination condition is specified, the combined fragment only documents the interactions that are executed if an unspecified termination condition is true. For the precise specification of requirements, it is imperative, however, that the termination conditions are explicitly documented. If a termination happens during the execution of a scenario, only the interaction in the termination fragment is executed—that is, the execution of the scenario ends after executing the interaction in the termination fragment. This happens even if there are further interactions specified in the sequence diagram after the termination fragment. These interactions are executed if the termination condition is not true during the execution of the scenario.

If a termination fragment does not contain an interaction, the scenario ends right after the termination condition is true. Figure 88 shows an example of the modeling a combined fragment of the type "break".

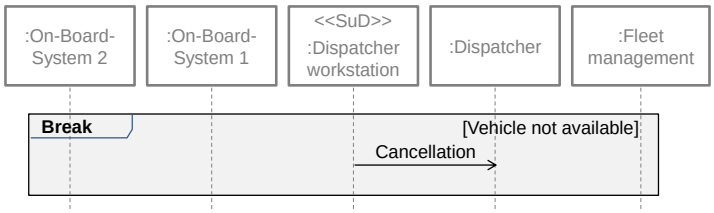


Figure 88: Modeling of a combined fragment of the type "break"

5.5.3 Nesting fragments

The use of combined fragments makes it possible to model several potential sequences of a scenario in a single sequence diagram. This is particularly true if combined fragments are nested. For example, the use of a single alternative fragment that includes three alternative interaction sequences models results in three possible executions of the scenario.

In the case of an optional fragment, at least two potential executions of the scenario are possible: one that occurs if the corresponding condition for the execution of the interaction in the optional fragment is true, and another if the condition is false.

If one alternative within a combined fragment of the type "alternative" itself contains a combined fragment of the type "optional", two potential sequences of the scenario are possible with regard to the alternative interaction. In a similar way, this also applies to the nesting of other types of fragments. Sequence diagrams that contain such combined fragments therefore model several potential sequences of the corresponding scenario.

In this way, sequence diagrams can model related main, alternative, and exception scenarios (termination scenarios) in an understandable way. In this case, main, alternative, and exception scenarios are specified through a corresponding control flow of the scenario. Figure 89 shows an example of the modeling of combined nested fragments.

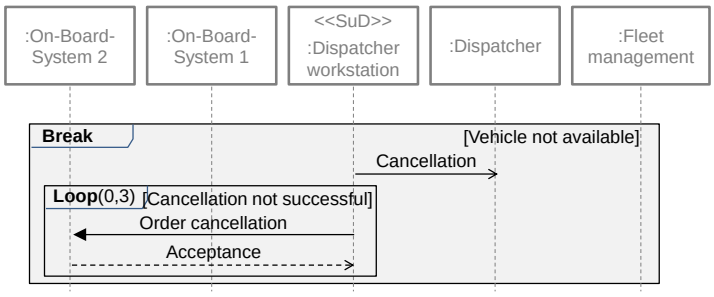


Figure 89: Modeling of combined nested fragments

5.5.4 Modeling assumptions of a scenario

Scenarios are typically based on a number of assumptions whose validity is assumed so that the scenario can actually be executed in the way it is modeled. If scenarios are modeled in sequence diagrams, the assumptions can be specified as textual annotations that are linked to the related model elements within the scenario. Figure 90 shows a simple example of the modeling of assumptions on which a scenario is based.

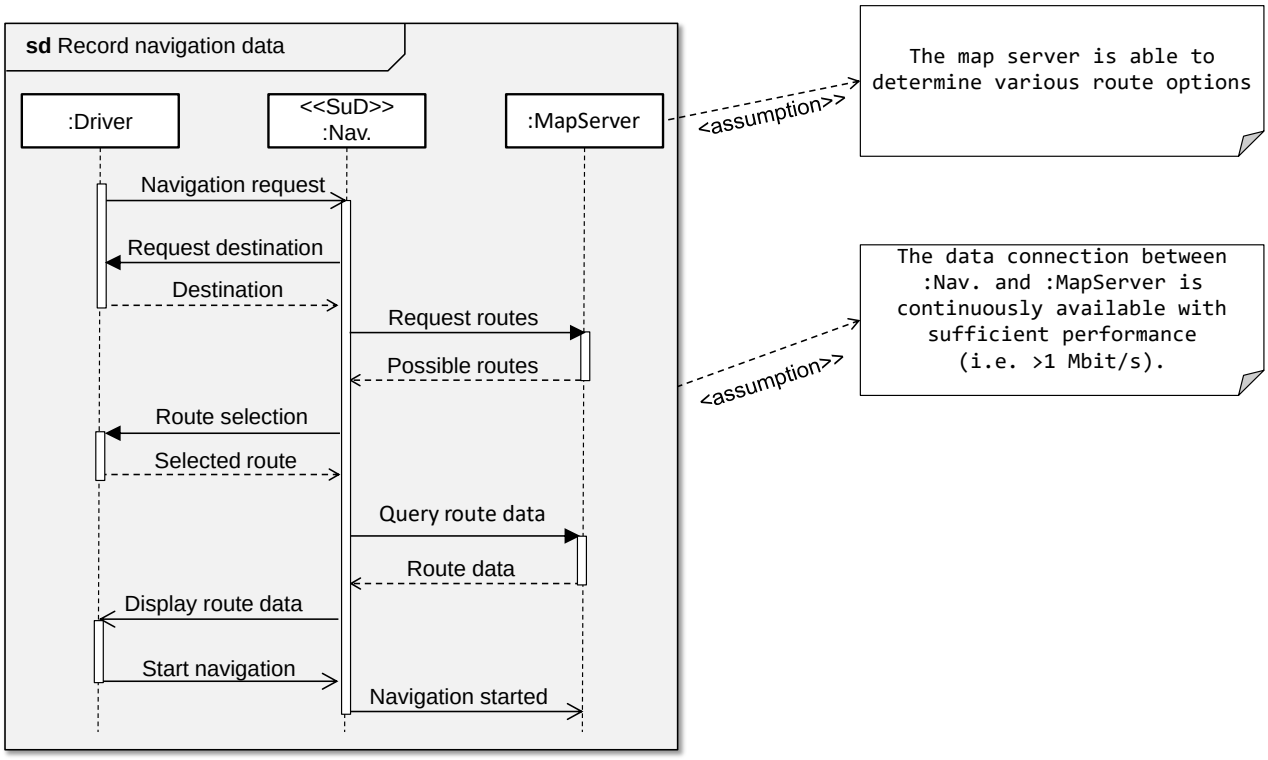


Figure 90: Modeling of assumptions for a scenario

The relationship between model elements of the sequence diagram and the associated assumptions is shown via a directed dependency relationship with the stereotype <<assumption>> (cf. Section 1.8). As shown in the figure, the assumptions can relate to the entire scenario or to single model elements within the scenario. The statement of such an assumption is, for example, that the scenario can only be completed successfully if ":MapServer" satisfies the related assumption.

This allows the exclusion of exception cases that do not contribute to the general understanding of the scenario, for example.

5.6 Scenario modeling with communication diagrams

Figure 91 shows the model elements of UML communication diagrams which are used for modeling scenarios. Communication diagrams also have an outer **frame** which contains the name of the scenario modeled by the communication diagram in a register at the top left.

The **name of the scenario** typically has the keyword "cm" as a prefix, indicating that the scenario is modeled by a communication diagram. A **lifeline** represents one instance of an actor within the scenario. The naming of the lifeline follows the pattern instance `name:type` name (e.g., `Peter:Driver`). A direct **message exchange** between two instances within the scenario is modeled by a connecting line between these instances in the communication diagram.

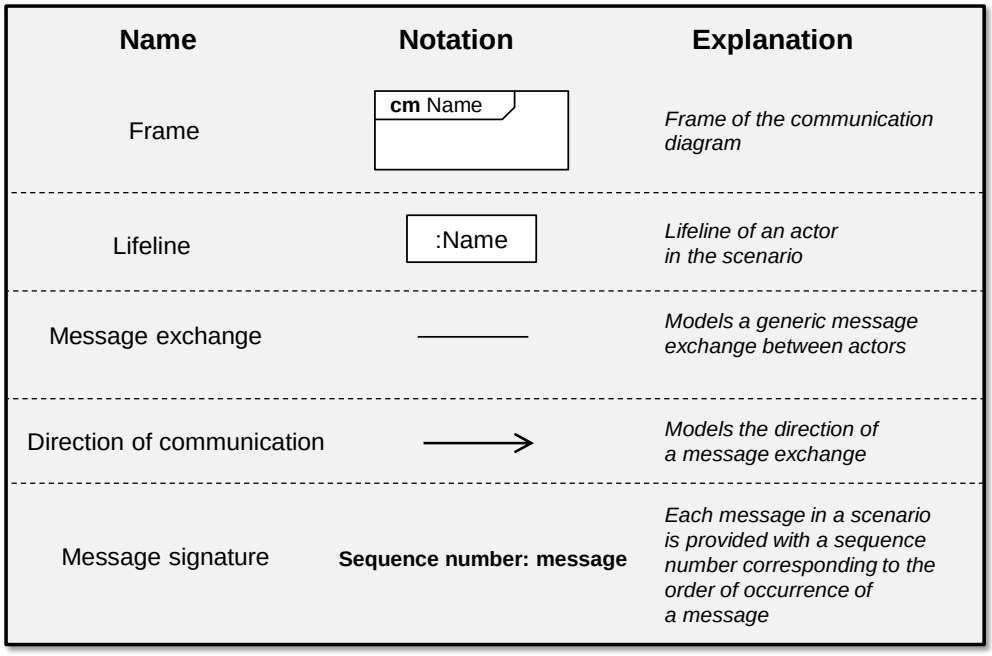


Figure 91: Model elements of communication diagrams for modeling scenarios

Each message that is exchanged between instances within the scenario is annotated with a **message signature** at the corresponding connecting line. The message signature consists of the actual **message** and the **sequence number** of the message exchange in the scenario. The **direction of communication** of a message is indicated by an arrow.

5.7 Examples of typical diagrams in the scenario view

With the help of various types of combined fragments, we can model complex interactions between actors and between actors and the system under development. Table 4 summarizes typical uses of combined fragments in scenario modeling as well as the consideration of scenarios within use cases.

Scenario level	Scenarios at the use case level	Fragment
Modeling of alternative sequences of messages between communication partners	Modeling of alternative extend relationships between use cases at an extension point	Alt
Modeling of optional messages between communication partners	Modeling of individual extend relationships between use cases that do not consider exception handling	Opt
Abstraction of a combined sequence of messages, e.g., for controlling complexity and improving readability	Modeling of include relationships between use cases	Ref

Scenario level	Scenarios at the use case level	Fragment
Modeling of repetitions of messages between communication partners within scenarios depending on conditions	—	Loop
Modeling of exception handling in scenarios	Exception handling via extend relationships between use cases	Break

Table 4: Typical uses of combined fragments in modeling scenarios

This section illustrates the use of the above types of combined fragments in the context of scenario modeling based on typical excerpts from the scenario view of a dispatcher's workstation in transport management.

5.7.1 Modeling scenarios using sequence diagrams

Figure 92 and Figure 93 show an excerpt from the scenario view for a dispatcher's workstation in the form of two UML/SysML sequence diagrams. The sequence diagram shown in Figure 92 illustrates the scenario "Provide replacement vehicle", which models the interaction between the instances: `On-Board System 2`, `:On-Board System 1`, `:Dispatcher Workstation`, `:Dispatcher`, `:Fleet Management` and: `Order acceptance`.

These interactions have to take place so that a replacement vehicle can be provided. The dispatcher workstation represents the software system under development; the other communication partners in the scenario are instances of actors in the system context.

The scenario shown uses both basic model elements for scenario modeling with UML/SysML sequence diagrams and advanced model elements: two repetition fragments (keyword "loop") and a termination fragment (keyword "break"). The first repetition fragment models that the dispatcher workstation attempts to send the transport documents a maximum of three times. After the dispatcher workstation sends the transport documents, it waits for the acceptance by the on-board system of the replacement vehicle (i.e., a synchronous message). This interaction is executed as long as the condition "Acceptance not successful" is true.

If the condition is false when entering the combined fragment, the corresponding interaction in the combined fragment is no longer executed. The dispatcher workstation sends the asynchronous message "Vehicle selection" to the dispatcher.

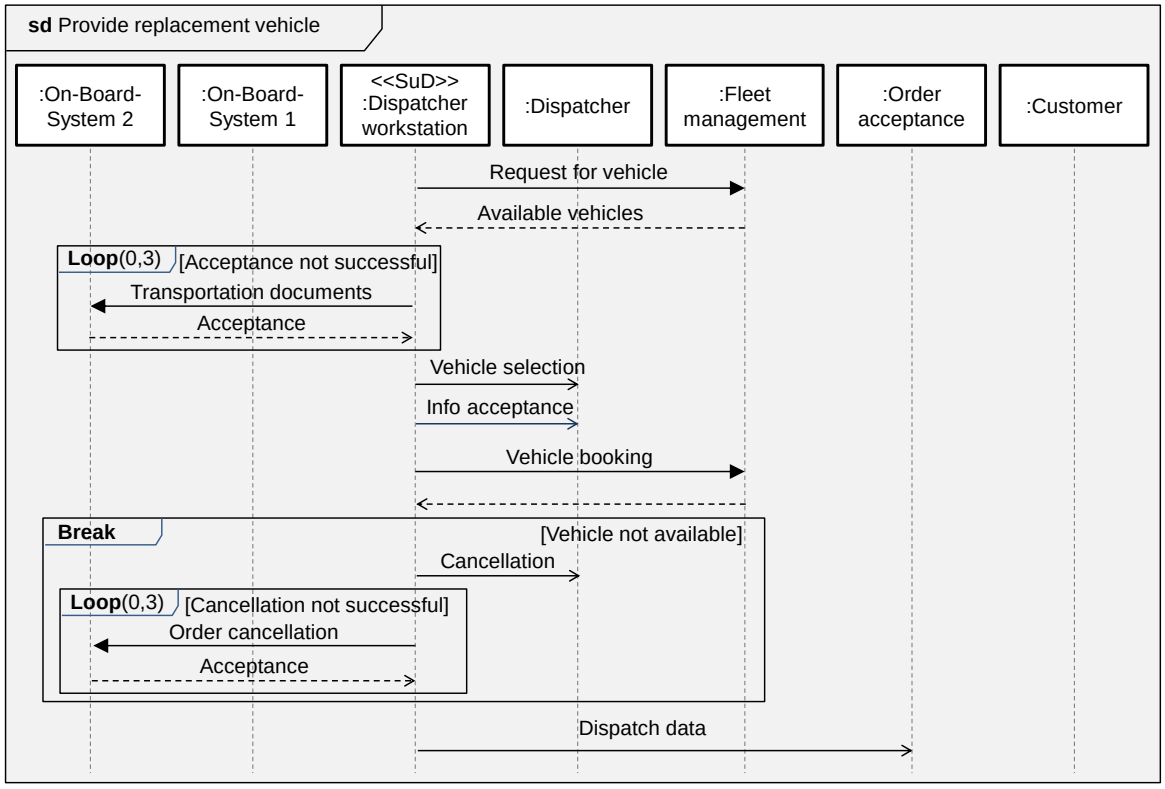


Figure 92: Example of a scenario modeled through a sequence diagram

The termination fragment models that if the condition "Vehicle not available" is true, an asynchronous message is sent from the dispatcher workstation to the dispatcher. It also models the interaction to cancel the order between the dispatcher workstation and the on-board system, which is repeated a maximum of three times.

If the condition "Cancelation not successful" is true when entering this fragment (i.e., the cancelation was unsuccessful), the interaction within the repetition fragment is no longer executed. If the termination fragment was entered, the scenario terminates after the execution of the interaction within the termination fragment, meaning that the asynchronous message "Dispatch data" is no longer sent from the dispatcher workstation to the order acceptance.

Figure 93 illustrates the sequence diagram that models the scenario "Replacement order for transport damage". It shows the interaction between the instances `:On-Board System 2`, `:On-Board System 1`, `:Dispatcher Workstation`, `:Dispatcher`, `:Fleet Management`, `:Order Acceptance` and `Customer`, which has to take place so that a substitute delivery can be notified in the case of transport damage. Various advanced model elements of scenario modeling with sequence diagrams were used to model the scenario "Replacement order for transport damage".

For example, the alternative fragment at the beginning models that if the electronic message for transport damage occurs, the transport damage message is sent from the on-board system of the vehicle to the dispatcher workstation which then sends a message containing the damage information to the dispatcher.

Alternatively, the transport damage message can reach the dispatcher in other ways. In this case, the message about damage that has occurred is sent directly to the dispatcher in another way (→ Found message). The dispatcher then has to enter the necessary damage information for further processing via the dispatcher workstation.

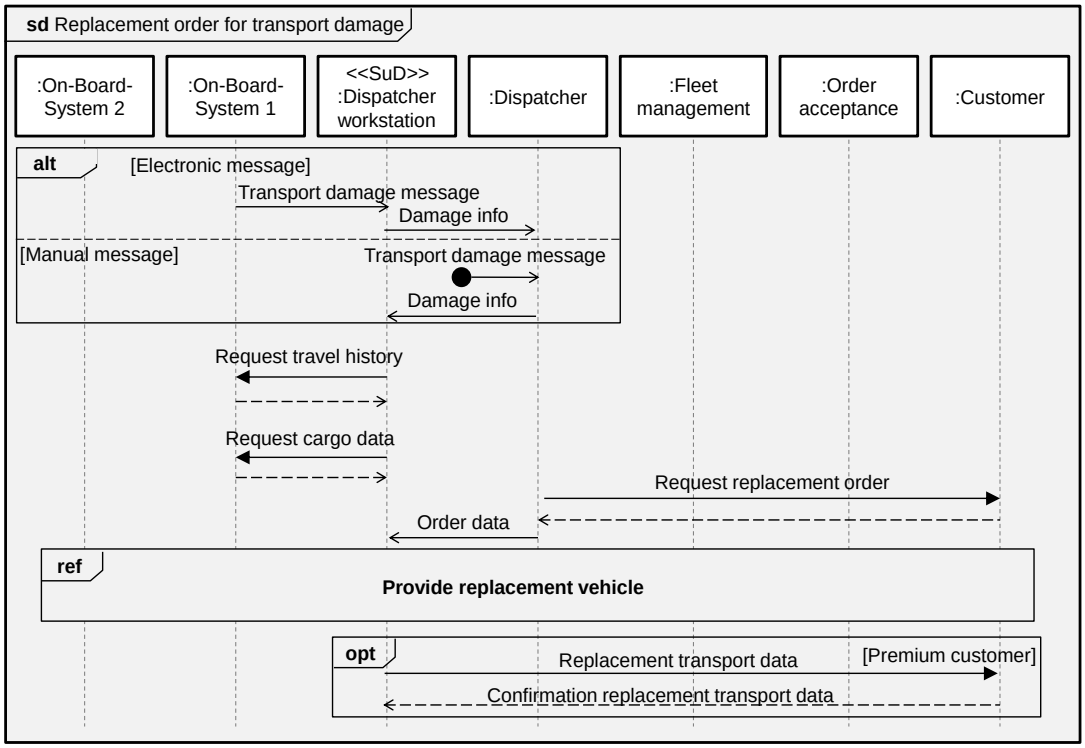


Figure 93: Example of a scenario modeled using a sequence diagram

The reference fragment in the lower part of the sequence diagram documents that at this position in the sequence of the scenario, the interaction of the scenario "Provide replacement vehicle" (Figure 92) is included. The optional fragment at the end of the sequence diagram describes that, within the scenario, the dispatcher workstation sends a message with the replacement transport data to the customer and waits for a confirmation. However, this only occurs if the condition "Premium customer" is true, that is, if the transport customer is a premium customer. If this is not the case, the scenario terminates at the end of the interactions of the included scenario "Provide replacement vehicle".

5.7.2 Modeling Scenarios using Communication Diagrams

Figure 94 shows an excerpt from the scenario view for a dispatcher's workstation in the form of a UML communication diagram which models the scenario "Provide replacement vehicle" (see also Figure 92). It is obvious from the figure that communication diagrams are hardly suitable for modeling complex interaction-based behavior of scenarios since this diagram type does not have model elements that allow the modeling of "optional" or "alternative" interaction sequences of scenarios.

Moreover, communication diagrams do not have model elements that allow the abstraction of parts of an interaction sequence by modeling these interactions in a different diagram to which the parent diagram can reference.

Nevertheless, communication diagrams are advantageous if the focus is on the bilateral exchange of messages between instances of a scenario.

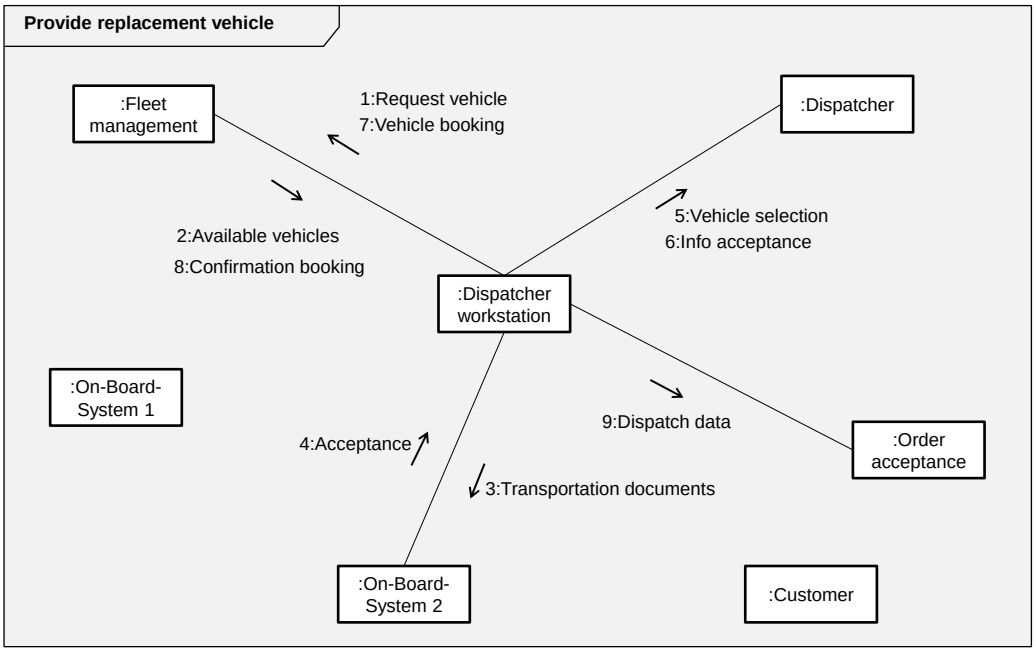


Figure 94: Example of a scenario modeled using a communication diagram

If the requirements engineer wants to model a scenario which does focus on this bilateral exchange of messages, the use of this type of diagram is beneficial. If necessary, sequence diagrams may be used in addition to a communication diagram to model scenarios. This might be the case, for example, if the focus is on modeling the properties of the bilateral interfaces (human–machine and machine–machine) between the system under development and the instances of actors.

5.8 Further reading

Types of scenarios and their documentation

- Rolland, C.; Achour, C.; Cauvet, C.; Ralyté, J.; Sutcliffe, A.; Maiden, N.; Jarke, M.; Haumer, P.; Pohl, K.; Dubois, E.; Heymans, P.: A Proposal for a Scenario Classification Framework. In: Requirements Engineering, 3 (1998) 1, S.23–47
- Jacobson, I.; Christerson, M.; Jonsson, P.; Oevergaard, G.: Object Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Reading, 1992.

Scenario modeling in requirements engineering

- Pohl, K.: Requirements Engineering – Fundaments, Principles, Techniques. Springer, 2010.

Modeling of sequence and communication diagrams

- Object Management Group: OMG Systems Modeling Language (OMG SysML) Language Specification v1.2. OMG Document Number: formal/2010-06-02.
- Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, Language Specification v2.41.
- Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, 2004.

6 Glossary

This glossary is partly based ofn: Glinz, M.: A Glossary of Requirements Engineering Terminology. Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, https://www.ireb.org/en/downloads/#cpre-glossary or https://www.ireb.org/en/cpre/glossary/.

Action	In requirements modeling, a ↑function of the ↑system that cannot be decomposed any further from a ↑requirements perspective; a primitive ↑function.
Activity	In requirements modeling, a complex ↑function of the system under development that, from a requirements perspective, can be decomposed into further ↑activities or ↑actions.
Activity diagram	A diagram type in UML which models the flow of ↑actions in a ↑system or in a ↑component, including ↑data flows and areas of responsibility where necessary.
Actor	A person or a technical system in the context of a system which interacts with the system under development.
Aggregation	Special type of association for modeling part/whole relationships.
Alternative scenario	A ↑scenario which describes an alternative sequence of ↑interactions, related to the basic scenario, for achieving the technical added value.
Association	A relationship between model elements—for example, a relationship between ↑classes in a ↑class diagram.
Attribute	A characteristic property of an ↑entity or an object. Attributes are defined on a type level, that is, entity types (ER diagrams) or classes (class diagram).
Main scenario	A scenario which, in relation to a specific outcome (e.g., a specific added value), describes the predominantly occurring sequence of interactions for achieving this result.
Class	Represents a set of ↑objects of the same kind by describing the structure of the objects, the ways they can be manipulated, and how they behave.
Class diagram	A diagrammatic representation of a ↑class model or a part of a class model.
Communication diagram	A diagram for modeling the behavior in the interaction-related ↑view which considers a logically related set of ↑interactions between objects and/or communication partners which focuses on the visualization of bilateral ↑interactions between communication partners. The causal order of ↑interactions is indicated here by sequence numbers.
Composition	Special type of ↑association for modeling part/whole relationships.

Context diagram	1. A diagrammatic representation of a ↑context model. 2. In ↑Structured Analysis, the context diagram is the root of the data flow diagram hierarchy.
Context view	A ↑requirements view which focuses on the demarcation of the ↑system boundary from the ↑context, that is, on the consideration of the ↑actors or neighboring systems of the ↑system under development and the interfaces between the system and these neighboring systems. In the context view, often only the ↑operational context of the system under development is modeled by ↑context diagrams.
Control flow	Temporal or logical sequence of, for example, ↑functions, ↑actions, or ↑activities.
Data flow	Representation of information (in a ↑data flow diagram or ↑activity diagram) that is exchanged between the ↑system context and/or ↑functions of the ↑system. (Data in motion, inputs and outputs of ↑functions).
Data flow diagram	A diagram modeling the ↑functionality of a ↑system or component using processes (also called activities), data stores, and data flows. Incoming data flows trigger processes which then consume the received data, transform it, read/write persistent data held in data stores, and then produce new data flows which may be intermediate results that trigger other processes or final results that leave the system.
Data type	Specification of a complex information structure for the definition of ↑attributes.
Diagram	Graphical description of a coherent set of properties of the object under consideration. Instance of a specific ↑diagram type.
Diagram type	Defines a class of "similar" ↑diagrams and is defined by a ↑modeling language.
Event	Timeless event that characterizes the occurrence of a condition, the termination of an ↑action or ↑activity, or the arrival of a ↑data flow or message.
Exception scenario	A ↑scenario describing a sequence of ↑interactions that must be executed if a defined exception event has occurred during operation of the ↑system. In requirements engineering, ↑exception scenarios are often specified complementary to the ↑main scenario and/or ↑alternative scenarios for the controlled treatment of scenarios.
Function (of a system)	In requirements models, a generic term for use cases, ↑activities, or ↑actions that are required in a requirements specification for the ↑system.
Generalization	A concept for the abstraction of common properties such as ↑classes, in which the common properties are merged into a generalized concept and the differences are depicted in respective specialized concepts.
Instance scenario	A ↑scenario in which communication partners and interactions are considered at the instance level.

Interaction	An interaction is a flow of tangible (e.g., money) or intangible things (e.g., information) between two or more communication partners.
Interaction-based view	The interaction-based view is a special ↑dynamic view of the ↑requirements of the ↑system under development in which the behavior is observed through interactions between communication partners.
Model	Abstracting image of an existing reality or an example for a planned reality (e.g., a system).
Model element	An atomic component of a diagram or a textual supplement to the requirements model. A model element typically represents a single requirement for the system.
Modeling construct	An atomic component of a diagram type (e.g., class, association, state, or state transition).
Modeling language	A ↑language for expressing ↑models of a certain type. May be textual, graphic, symbolic, or a combination thereof.
Object	An occurrence/instance of a class.
Operational context	The part of the ↑system context with which the ↑system has a functional interaction during operation—for example, users, other systems, technical or physical processes, or business processes.
Pragmatic quality	Extent to which a ↑diagram/↑model serves its intended purpose in terms of the adequacy of abstraction.
Pragmatics	Part of the definition of a ↑modeling language which describes the intended use and possibly also describes the form and specific purpose of abstraction in order to fulfill the intended use as well as possible.
Process flow	*See* ↑Control flow
Requirements view	Defines, for reasons of complexity control, a specific abstraction of the requirements of a system in which only certain facts (e.g., ↑states and ↑state transitions of the system under development) have been considered and others have deliberately not been considered. Typically, the different views of the requirements can be combined into an overall model of the requirements.
Requirements model	A ↑model that has been created with the purpose of specifying ↑requirements. Consists of diagrams of various requirements views and textual additions.
Role	Designation of a class from the perspective of the other ↑class for an ↑association.
Scenario	An ↑interaction between communication partners (often between the ↑system under development and ↑actors in the system context) that leads to a desired (or possibly unwanted) result. In requirements engineering, the added value for an ↑actor in the system context is often seen as an essential result of a ↑scenario.

Semantics	Part of the definition of a modeling language; defines the general meaning of the notation elements (i.e., generally → What is the meaning of a class in a class diagram? Not → What is the meaning of the class "customer" in the class diagram?).
Semantic quality	Extent to which a ↑diagram/↑model reflects the specific view of the object under observation correctly and completely.
Sequence diagram	A diagram type in ↑UML which models the interactions between a selected set of objects and/or ↑actors in the sequential order in which those interactions occur.
Signal	An ↑event in or outside the system which is relevant to the ↑system under development.
State	A state is a summary of certain conditions that apply during a time interval for a ↑ system or subsystem.
State diagram	The graphical representation of a state machine.
State machine	Through a summary of ↑states and ↑transitions between these states, a state machine describes the behavior or part of the behavior of the object considered (e.g., an ↑actor, a ↑function, a ↑use case, or the ↑system).
State machine diagram	*See* ↑State diagram
Statechart	*See* State machine
Syntactic quality	Extent to which the ↑diagram/↑model satisfies the underlying syntactic rules.
Syntax	Part of the definition of a ↑modeling language that defines the way the available notation elements in the modeling language can be combined (the grammar).
System	Entity with defined borders and an interface through which the entity interacts with its environment (context). Typically consists of a set of related components.
System boundary	Demarcates the ↑system from its context (e.g., via responsibilities and exclusions).
System context	Aspects outside the system that are relevant for the definition of the ↑requirements of a system and their relationships to each other and to the system under development. The system context includes the ↑operational context, that is, the part of the environment with which the operational system is in a functional interaction.
System environment	See Operational context
System under development	The system considered in the context of requirements engineering or requirements modelling.
System under study	A system to be considered or analyzed in the context of system analysis. Not necessarily the object of development.
Transition	A change from one ↑state to another initiated by a trigger.
Trigger	The processing of a signal as an actuator for a transition.
Type scenario	A scenario in which communication partners and interactions (↑) are considered at the type level. Scenarios (↑) within a use

	case specification are often at the type level, that is, they consider types of communication partners and types of interactions.
Use case	A description of the possible interaction between an actor and the system which, when executed, yields an added value.
Use case diagram	A diagram type of UML which allows the modeling of ↑actors and ↑use cases of a system. The line between actor and use case represents the ↑system boundary. Use case specification: The textual description of a use case.
Use case scenario	A possible sequence (trace) of the interactions within a use case. The possible sequences are represented by the main, alternative, and exception scenarios of the use case.
View	An abstract representation of the ↑system under development, consisting of one or more ↑diagrams (with textual additions). Views can be disjoint or overlapping. Deliberate overlaps are applied for quality assurance of the models (to produce consistency by viewing the system from several perspectives).

7 List of Abbreviations

AD	Activity diagram
BPMN	Business Process Modeling Notation
CM	Communication diagram
CPRE	Certified Professional for Requirements Engineering
CRM	Customer relationship management
DFD	Data flow diagram
EPC	Event-driven process chain
ER	Entity relationship
FMC	Fundamental modeling concepts
IREB	International Requirements Engineering Board
ISO	International Organization for Standardization
IT	Information technology
ITU	International Telecommunication Union
OMG	Object Management Group
RE	Requirements engineering
SA	Structured Analysis
SD	Sequence diagram
SuD	System under development
SuS	System under development
SysML	System Modeling Language
UML	Unified Modeling Language

8 References

[Balz2011] Balzert, H.: Lehrbuch der Objektmodellierung – Analyse und Entwurf mit der UML 2, Spektrum Akademischer Verlag, Heidelberg 2011. (in German).

[BDH2012] Broy, M.; Damm, W.; Henkler, S.; Pohl, K.; Vogelsang, A.; Weyer, T.: Introduction to the SPES Modeling Framework. In: Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M.: Model-Based Engineering of Embedded Systems, Springer, Heidelberg 2012.

[Caro1995] Carroll, J. M.: The Scenario Perspective on System Development. In: J. M. Caroll (Hrsg.): Scenario-Based Design – Envisioning Work and Technology in System Development, Wiley, New York, 1995, S. 1–17.

[Chen1976] Chen, P.: The Entity-Relationship Model: Towards a Unified View of Data, ACM Transactions on Database Systems, 1976.

[CoNM1996] Coad, P.; D. North, D.; Mayfield, M.: Object Models: Strategies, Patterns, and Applications, Prentice Hall, 1996.

[Cock2000] Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Longman, Amsterdam 2000.

[Cohn2002] Cohn, M.: User Stories Applied: For Agile Software Development, Addison Wesley, 2002.

[DaLF1993] Dardenne, A.; Van Lamsweerde, A.; Fickas, S.: Goal-Directed Requirements Acquisition. Science of Computer Programming, Vol. 20, No. 1–2, Elsevier Science, Amsterdam, 1993, p. 3–50.

[DaTW2012] Daun, M.; Tenbergen, B.; Weyer, T.: Requirements Viewpoint. In: Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M.: Model-Based Engineering of Embedded Systems, Springer, Heidelberg 2012.

[Davi1993] Davis, A. M.: Software Requirements – Objects, Functions, States. 2nd Edition, Prentice Hall, Englewood Cliffs, New Jersey 1993.

[DeMa1979] DeMarco, T.: Structured Analysis and System Specification, Yourdon Press, Pentice Hall, 1979

[Fowl1996] Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley, Reading, MA 1996.

[GaJV1996] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA 1994.

[GaSa1977] Gane, C.; Sarson, T.: Structured Systems Analysis – Tools & Techniques. Improved System Technologies, New York 1977.

[Glin2011] Glinz, M.: A Glossary of Requirements Engineering Terminology. Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version 1.1, May 2011.

[HaCa1993] Hammer, M., Champy, J.: Reengineering the Corporation: A Manifesto for Business Revolution, Harper Business Essentials, 1993.

[HaHP2001] Hatley, D., Hruschka, P., Pirbhai, I.: A Process for System Architecture and Requirements Engineering, Dorset House, 2001.

[Hare1987] Harel, D.: Statecharts – A Visual Formalism for Complex Systems. Science of Computer Programming, Vol. 8, No. 3, 1987, p. 231–274.

[HKDW2012] Hilbrich, R.; Van Kampenhout, J. R.; Daun, M.; Weyer, T.: Modeling Quality Aspects: Real-Time. In: Pohl, K.; Hönninger, H.; Achatz, R.; Broy, M.: Model-Based Engineering of Embedded Systems, Springer, Heidelberg 2012.

[IEEE1471] IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Standard 1471–2000.

[ISO25010] ISO/IEC/IEEE Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation. ISO/IEC/IEEE Standard 25010:2011.

[ISO26702] ISO/IEC/IEEE Systems and Software Engineering – Application and Management of the Systems Engineering Process. ISO/IEC/IEEE Standard 26702:2005.

[ISO29148] ISO/IEC/IEEE Systems and Software Engineering – Life Cycle Processes – Requirements Engineering. ISO/IEC/IEEE Standard 29148:2011.

[ISO42010] ISO/IEC/IEEE Systems and Software Engineering – Architecture description. ISO/IEC/IEEE Standard 42010:2011.

[ITU2004] International Telecommunication Union: ITU-T Z.120 Message Sequence Chart (MSC), 2004.

[JCJO1992] Jacobson, I.; Christerson, M.; Jonsson, P.; Oevergaard, G.: Object Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Reading, 1992.

[LaSi1987] Larkin, J. H.; Simon, H. A.: Why a diagram is (sometimes) worth ten thousand words. In: Cognitive Science, Vol. 11, 65–99.

[LiSS1997] Lindland, O. I.; Sindre, G.; Sølverg, A.: Understanding Quality in Conceptual Modeling. IEEE Software, Vol. 22, No. 2, IEEE Press, 1994, 42–49.

[Mart1989] Martin, J.: Information Engineering, Book I – Introduction. Prentice Hall, Englewood Cliffs 1989.

[McPa1984] McMenamin, S. M.; Palmer, J. F.: Essential Systems Analysis. Prentice Hall, London 1984.

[Nuse2001] Nuseibeh, B.: Weaving Together Requirements and Architectures. IEEE Computer, Vol. 34, No. 3, IEEE Computer Society, Los Alamitos, 2001, 115–117.

[OMG2012] OMG Object Constraint Language (OCL); Version 2.3.1; January 2012 https://www.omg.org/spec/OCL/2.3.1/PDF. Last visited April 2024.

[OMG2010a] Object Management Group: OMG Systems Modeling Language (OMG SysML) Language Specification v1.2. OMG Document Number: formal/2010-06-02.

[OMG2010b] Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, Language Specification v2.41.

[OMG2010c Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure, Language Specification v2.41.

[OMG2011] Object Management Group: OMG Business Process Model and Notation (OMG UML), Language Specification v2.0.

[Pohl2010] Pohl, K.: Requirements Engineering – Fundaments, Principles, Techniques. Springer, Heidelberg 2010.

[RuJB2004] Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, Reading, MA 2004.

[BoRJ2005] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide. Addision Wesley, Reading, MA 2005.

[PoRu2011] Pohl, K.; Rupp, C.: Requirements Engineering Fundamentals – A Study Guide for the Certified Professional for Requirements Engineering Exam – Foundation Level – IREB compliant, RookyNook Computing, 2011.

[Pott1995] Potts, C.: Using Schematic Scenarios to Understand User Needs. In: Proceedings of the ACM Symposium on Designing Interactive Systems – Processes, Practices, Methods and Techniques (DIS'95), ACM, New York, 1995, S. 247-266.

[RaJa2001] B. Ramesh, M. Jarke: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, Vol. 27, No. 1, IEEE Press, 2001, S. 58-93.

[RiWe2007] Rinke, T.; Weyer, T.: Defining Reference Models for Modeling Qualities – How Requirements Engineering Techniques can Help. In: Proc. of the 13th Intl. Working Conf. on Requirements Engineering – Foundation for Software Quality, Lecture Notes in Computer Science, 4542, Springer 2007.

[RoRo2006] Robertson, S.; Robertson, J.: Mastering the Requirements Process. 2nd edition, Addison-Wesley, Amsterdam, 2006.

[RAC1998] Rolland, C.; Achour, C.; Cauvet, C.; Ralyté, J.; Sutcliffe, A.; Maiden, N.; Jarke, M.; Haumer, P.; Pohl, K.; Dubois, E.; Heymans, P.: A Proposal for a Scenario Classification Framework. In: Requirements Engineering, 3 (1998) 1, S. 23-47.

[RoSc1977] Ross, D. T.; Schoman, K.E.: Structured Analysis for Requirements Definition. IEEE Transactions on Software Engineering, Vol. 3, No. 1, 1977, p. 6-15.

[Sche2000] Scheer, A.-W.: ARIS – Business Process Modeling. 3rd edition. Springer, Berlin 2000.

[ShMe1988] Shlaer, S.; Mellor, S.: Object-oriented Systems Analysis – Modeling the World in Data. Prentice Hall, Englewood Cliffs 1988.