WORKBOOK



## Software Requirements Specification (SRS)

ISPE01



Key learning objectives

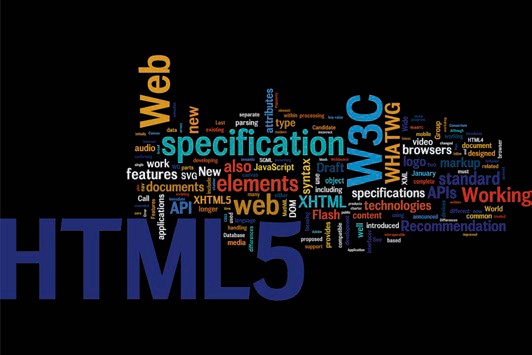
##### Introduction 9



All software projects require detailed specification of the characteristics that their systems or system components must offer. Viewed from an external perspective, this requires a functional-technical description of the system’s internal interactions with the processed business objects as well as its interfaces with its environment.

This course on “Software Requirements Specification” will explore the elements needed for a detailed specification, explain how to document them, and investigate the methodology and approaches used for specification. The focus is on commonly used elements and documentation formats for the specification of industrial IT systems.

[www.iubh.de](http://www.iubh.de/)



# Unit 1

## Introduction to Software Requirements Specification (SRS)

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... what SRS is and how it is used in software engineering.

... the typical system elements described in SRS and how to identify them.

... the individual elements defined in specification documents and the formats used.

DL-D-ISPE01-L01

1. Introduction to Software Requirements Specification (SRS)

### Introduction

In this unit, we will explore what is meant by software requirements specification and how it is used in software engineering, together with the elements commonly used in SRS and how to identify them. We will then take a look at typical structures and documentation formats.

Note

Module 1.1 is essentially a refresher of the “Introduction to industrial software engineering” course. Students who have already completed the module on

“Introduction to industrial software engineering” will already be familiar with most of the content.

### Fundamental principles and terminology of SRS

###### SRS versus requirements engineering

Software requirements specification documents the external requirements placed on a software system. Building on the outcome of the requirements engineering (RE) process, which defines, documents, verifies, and coordinates the conceptual requirements, SRS produces highly technical documentation of the system being designed.

Some of the specialist software engineering literature defines technical specification as the outcome of requirements engineering and makes no distinction between requirements engineering and SRS. In this course, the term “requirements engineering” (RE) is used to refer to the activities, methods, and techniques for identifying, documenting, verifying, and coordinating the conceptual requirements, while SRS refers to the documentation of detailed technical requirements. SRS builds on requirements engineering and documents the requirements in greater depth. The investigative techniques and testing methods used are the same for both RE and SRS.

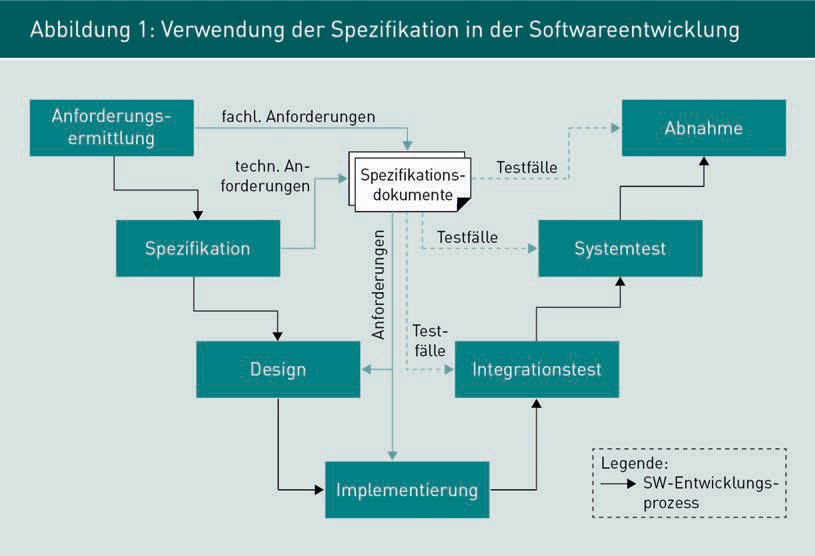
We believe this distinction to be relevant because RE focuses primarily on the situation-dependent selection and application of investigative, documentation, and testing techniques for conceptual requirements.

Introduction to software requirements specification (SRS)

Against this background of stable conceptual requirements and prompted by the need for precise technical specifications, the focus then shifts to finding the most suitable techniques for specifying selected conceptual/technical requirements. The RE requirements are elaborated and refined at a technical level until they are sufficiently detailed to allow the development team to begin its design work.

###### Use of SRS in software development

Figure 1 illustrates how SRS documents are used over the course of software development. During the SRS process, the identified conceptual requirements are elaborated and refined to include technical requirements. The resultant conceptual/technical specification serves as the basis for designing and later implementing the system. The specification is also used to prepare test cases for testing the software system at various levels to ensure compliance with the requirements.



Since the specification forms the basis for system design and implementation as well as test cases, any errors in the specification or misunderstandings can have wide-ranging implications for the entire project. Software engineering offers a wide selection of visualization techniques (such as software models and specification languages) and tools (such as modeling tools) for defining the specific elements of a software system and ensuring that the requirements are clearly and measurably specified.

###### Distinguishing between specification and design

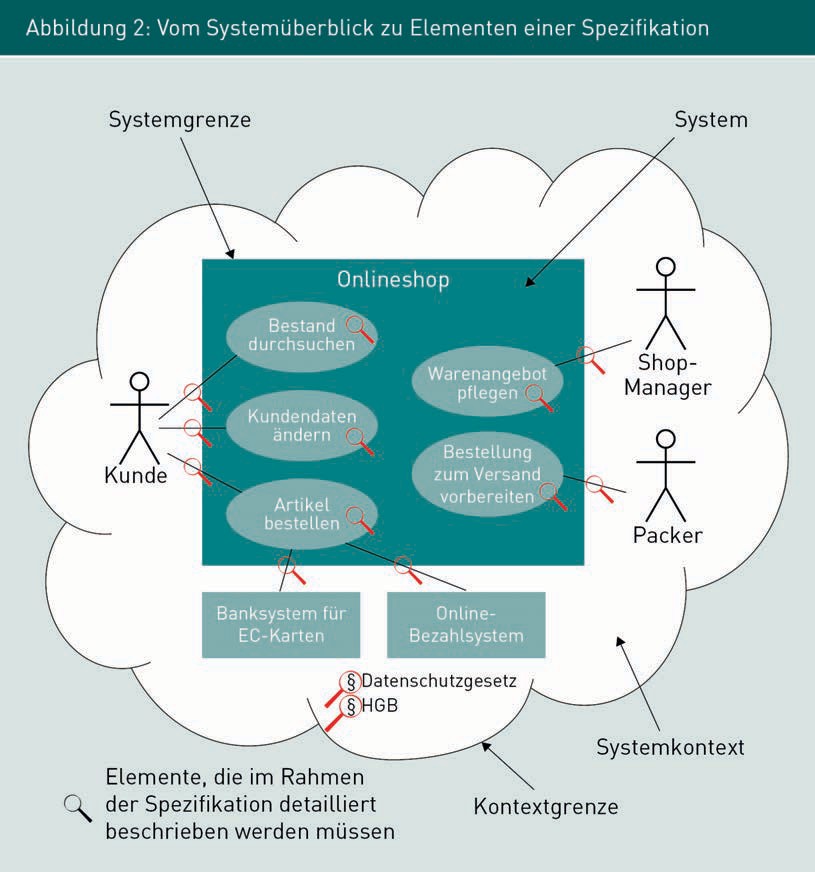
From a conceptual perspective, the system specification provides a detailed technical framework for design decisions. It is important to understand that no decisions about the internal design of the system are made in the SRS; it merely describes those system properties which are externally visible. From an SRS perspective, you could say that the system is a black box with an unknown internal structure.

### Elements and structure of SRS

###### Identifying which system elements to specify

As previously mentioned, the specification defines what the system needs to be able to do. In other words, it describes the system’s functional behavior as viewed from the outside. Based on the outcome of the requirements engineering process, it may be useful to model the system context in the form of a UML use case diagram to help identify the required specification elements. An example of a UML use case diagram is shown in Figure 2, whereby the elements considered particularly relevant for the specification are labeled with red magnifying glasses.

Introduction to software requirements specification (SRS)



The system boundary marks out the design zone. All elements outside of these system boundaries will normally be outside the project’s sphere of influence. All use cases within the system are relevant to the specification. Defined conceptual and technical operations within a use case and the required rules, business objects, and system components must all be included in the specification.

Furthermore, each point where a communication relationship intersects with the system boundary requires an in-depth specification of the system interface. Communication relationships between human actors and the system (e.g. the relationship between “customer” and “change customer data” in Figure 2) usually take the form of a graphical user interface (GUI). If the actor is another system, a technical system interface is needed to enable the two systems to communicate with one another.

Where other legal, technical and organizational framework conditions apply in the system context, these must be carefully scrutinized to ascertain their specific influence on the system. Alongside functional requirements, framework conditions often include quality features which must be elaborated into measurable quality characteristics of the system during the specification process.

In a given project situation, therefore, denoting the system context as a UML use case diagram, allows us to identify and model the key elements of the specification.

###### Specification of use cases

While requirements engineering focuses on identifying and modeling business operations and the associated business objects, the purpose of SRS is to identify and define the conceptual system elements needed to satisfy the requirements highlighted by the use case. It assigns specific business functions to each conceptual system element and specifies their behavior in detail. Elements for specifying a system’s behavior include the following:

* Data model: A data model denotes the business objects processed by the system components and their relationships to one another. Examples: Filing a claim, an insurance application, client data.
* Specialist functions: A functional description of the tasks performed by the system or specified component. Examples: Algorithm for calculating premiums, cancelation procedure, signing a contract.
* Business rules: Business rules relate to a business object and must not be violated. Examples: The contract start date must precede the contract end date. The sum total of the shopping cart cannot be negative.

In Unit 3 we explore the specification of conceptual components in greater depth, while Unit 5 focuses on the specification of detailed data models.

###### Specification of graphical user interfaces (GUI)

The specification of graphical user interfaces helps the development team to design a GUI that is precisely tailored to the stakeholders’ needs. It should also enable the test team to design and implement actual test cases for the GUI. Specification of a GUI typically includes the following elements:

Introduction to software requirements specification (SRS)

* Content and structure of individual dialog boxes: Details concerning the type, size, position, color, and content of elements on a screen page, such as input fields, texts, buttons, and images.
* Data conversion and validation: Specification of rules for checking input fields for technical plausibility.
* Dialog flow: Specification of user prompting through the interface depending on the user’s data entries and actions.

Unit 2 explores the techniques and elements used in the specification of GUIs in greater depth.

###### Specification of technical system interfaces

As previously mentioned, as well as system interfaces to human users, there are also system interfaces to other IT systems. As the system being designed will communicate externally via technical interfaces rather than via GUIs, these technical interfaces must also be specified. The following characteristics of interfaces may be defined:

* Purpose of the interface on a conceptual level. Examples: Transfer share prices into the fund management system, validate address data
* The detailed response / technical communications protocol and rules used by the system to communicate with its environment. Example: HTTP, FTP.
* The data structure of messages exchanged at the interface. Example: XML, CSV.

In Unit 4 we explore various techniques for specifying system behavior at technical interfaces. This is followed by a detailed specification of conceptual data models in Unit 5. Unit 6 looks at the specification of actual data interfaces using the exchange format XML.

###### Specification of quality characteristics

As well as specifying system behavior, data structures, and interfaces, the SRS also defines the quality characteristics required of the system. Standard ISO 9126 defines the term “software quality” as follows: “The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.” “Good usability” or permanent “data protection” or “constant” availability of the system are often prescribed as required quality characteristics.

Software quality: The totality of features and characteristics of a software product that bear on its ability to satisfy

stated or implied

needs.

When preparing a detailed specification of quality characteristics, measurable quality criteria are derived from the general requirements outlined above. These precisely formulated quality characteristics will have a major influence on the system’s architecture: Will the system be used simultaneously by five or 500,000 people, or must the system be available 24/7 or only from 9.00 am to 4.00 pm on weekdays? These types of quality characteristics tend to have a far greater influence on technical implementation than adding individual functions to the functional scope.

The testability of quality criteria is also a key consideration. We have already learned that the SRS is used to prepare precise test cases. Quality characteristics must therefore be defined in a way that enables their compliance to be tested.

Unit 7 explores the methodical, structured identification, and specification of quality characteristics in depth.

### Structure and documentation of SRS

###### Structure of specification documents

Like conceptual requirement documents, there is no generally accepted structure for detailed specification documents. The format is governed by the system type, the individual system elements being specified, the framework conditions of the project, and the guidelines and requirements of participating companies. Whereas the document structure in requirements engineering tends to focus more on the early phases of the software process, SRS documents the required system properties in depth from a conceptual/technical perspective.

A specification document should typically include the following sub-headings:

1. Meta-information about the document

The meta-information contains all the data needed to make the document usable and readable. Meta-data does not contain any information about the system itself but helps guide the reader through the document and includes the document’s current status and contributing individuals. Specific examples include:

* Directories, key terminology, notations used
* Brief description of the content

Introduction to the specification of software systems

* + Notes on the structure
  + References to versions, author(s), date, current status

1. Introduction

The introduction sets out the project’s objectives and provides a very high-level overview, so that the system can be positioned within the context of the organization’s application landscape. Specifically, the introduction may include:

* The system’s purpose or the aim of the project
* Groups of individuals with whom the system will interact.

1. System overview

The system overview is equivalent to the refined overview level in the conceptual requirements document. It provides readers with an overview of the system’s principal functions, technical interfaces, users, and positioning within the system landscape. Unlike the conceptual requirements, it may also include documentation about the system’s technical embedding in the application landscape, for example in the form of architectural drawings. The system overview should incorporate the following elements:

* Broad functional description of the system using data models, specialist functions, business rules, and quality characteristics
* Description of the system environment in the form of architectural drawings
* Overview of the functional operations and data formats at interfaces.

1. Conceptual system components (and technical components, if known)

This section contains detailed information about the system and its interfaces. Specific functions and interfaces are assigned to so-called system components. It may also define conceptual and/or technical system components and their interfaces, depending on the specific project and current level of knowledge. Specifically, it includes the following elements:

* Conceptual system components and their correlations
* Detailed conceptual description of the components, including
  + Business objects used and their lifecycle, as well as
  + Specialist-related functions and business rules
* Technical description of component interfaces
* Component-specific quality features and peripheral conditions.

1. Rules that must be observed (compliance).

This section includes all relevant guidelines and regulations that must be observed during preparation and use of the system. Requirements placed on the system or its environment are often derived from these rules, which include both statutory regulations and in-house requirements. Statutory regulations include:

* + The Federal Data Protection Act on the handling of personal data
  + The Principles of Data Access and Auditing of Digital Documents (GDPdU) and the Generally Accepted Principles of Computerized Accounting Systems (GoBS): Auditing, logging, storage, and exchange of digital documents such as invoices
  + Act on Control and Transparency in Business (KonTraG) and Minimum Requirements for Risk Management (MaRisk (BA)): Internal auditing access; early risk detection mechanisms

1. Appendix

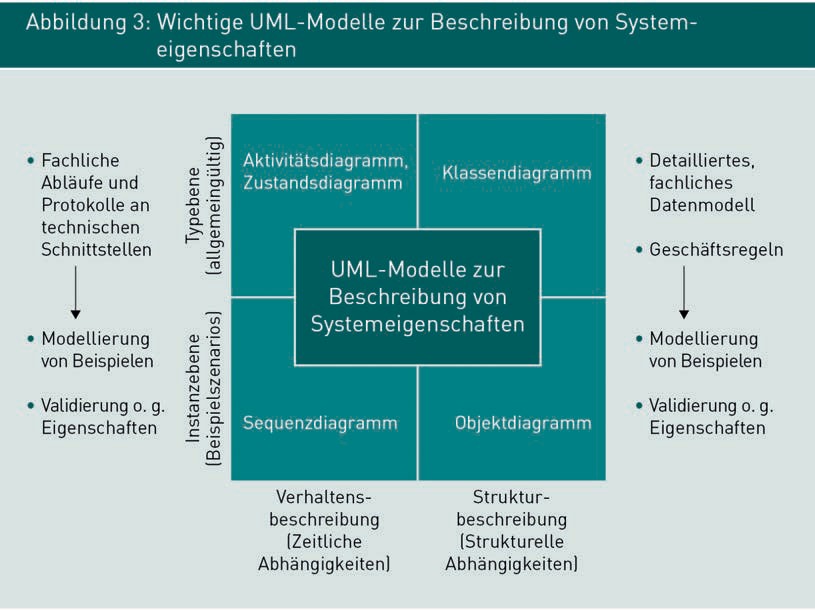
The Appendix at the end of the specification generally contains more in-depth information and technical documents in summarized form for clarity and readability. This information does not directly form part of the SRS but is intended to aid comprehension or has been extracted from the main body of the text for clarity. Typical information found in the Appendix includes:

* Glossary
* Specifications of surrounding systems or pre-existing components that are to be incorporated into the system
* Detailed technical data models, such as XML schema descriptions
* Reference to documents used (standards, regulations)
* Compliance information (rather than listing in a separate chapter) if of peripheral importance only.

###### Documentation formats

The documentation formats used for requirements engineering generally apply to the detailed technical specification as well. All system aspects can usually be specified in the form of texts and tables. However, practical experience has shown that structured descriptions are more suitable for higher levels of technical detail. The following overview explains how different UML diagram types can be used together in the specification of information systems. Figure 3 illustrates the correlation between different diagrams and their potential applications.

Introduction to the specification of software systems



We can see from the diagram that activity diagrams and state diagrams can be used to model operations. Possible system responses can be depicted in a universal fashion. Potential applications include the specification of GUIs, technical system interfaces, and operations within a system or component. In module 4.1, we will explore the UML sequence diagram. Sequence diagrams are used to model a very specific operation or activity diagram, whereby the system response is illustrated and specified using real-life examples.

Figure 3 also shows us that class diagrams can be used to specify the conceptual data model and business rules. The UML object diagram, presented in module 5.3, is used to denote a particular data record based on the structure prescribed in a class diagram. Like the sequence diagram, an object diagram can also be used to denote and illustrate facts using examples.

While activity and class diagrams are used for type-level modeling and therefore provide a general description of behavior (ACD) and structure (CLD), sequence and object diagrams are used for instance-level modeling. An object diagram is a defined version of a class diagram, while a sequence diagram is a version of an activity diagram.

Like requirements engineering, SRS employs use case diagrams to provide a system overview and identify individual system elements for specification, as illustrated in Figure 2. Other documentation formats used in specification include GUI prototypes for the specification of user interfaces, XML languages for the specification of data structures at technical system interfaces, decision tables for the specification of system behavior, and business rules used for the specification of data structures and system behaviors, depending on the situation.

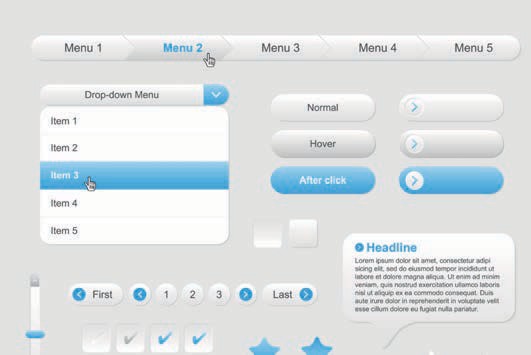
In the following units, we will consider which techniques are best-suited for specifying the typical elements of information systems.

Summary

The aim of SRS activities is to document the externally relevant requirements placed on a software system at a high technical level. Typical elements include the system’s externally visible behavior and its conceptual structure.

Conceptual functions are assigned to each system component, together with a detailed specification of its behavior. User interfaces (GUIs), technical system interfaces and quality characteristics must also be specified.

Since the specification provides the basis for implementation of the system (and the formulation of test cases), any errors or misinterpretations of the specification or may have wide-ranging implications for the entire project. The specification of information system therefore uses various different UML diagram types in combination with one another.



# Unit 2

## Specification of user interfaces (GUIs)

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... which elements a GUI specification should include

... the typical dialog elements and the types of data they are suitable for

... the information which must be specified for validation rules

... how to specify dialog flows in UML state diagrams

DL-D-ISPE01-L01

1. Specification of user interfaces (GUIs)

### Introduction

If user requirements and initial GUI prototypes have already been drawn up at the RE stage, the detailed SRS should specify all the relevant information relating GUIs to assist the development team. This unit explores which aspects you should consider in the specification of GUIs.

### Elements of a GUI specification

User interface The graphical user interface (GUI) is the system interface for communicating with

users.

The graphical user interface (GUI) refers to the parts of an application which are recognizable to human users. It is the interface where information is made available to and received from users in a format intelligible to humans.

Most industrial information systems have a graphical user interface. In other words, information is presented in a visual format to be seen by users. Alternatively, information may be given in an acoustic (e.g. as speech output) or haptic format (e.g. in Braille).

A typical user interface uses graphical symbols such as windows, menus, symbols, texts, and input elements to present information, and users interact with the software using a keyboard and pointing devices. Typical forms of user interaction include keyboard entries or clicking, holding, dragging, and releasing a pointing device. As touch screens and movement sensors are becoming ever more popular, gestures with one or more fingers or moving the input device within the space (e.g. shaking or tilting) are also supported.

The following account is limited to the specification of graphical user interfaces operated with a screen, keyboard, and pointing device as the most common type of GUI used in industrial information systems.

Industrial information systems, particularly operational systems, are used primarily to support business processes. The more industry-specific and individual an information system is, the more its GUI will be specifically modified and aligned with business- and organization-specific features. The system’s GUI helps to guide and navigate the user through the functional process.

Specification of user interfaces (GUIs)

Users can see the properties of domain objects and process them. They can also input decisions into the system where this is required to control operations in the business process.

Examples of scenarios where domain objects are used include:

* Entering travel details when booking a business trip
* Displaying and amending customer data in an online shop
* Capturing application data for a new household contents insurance policy.

Each of these scenarios relies on a dialog to guide the user through the system and assist with task processing. A dialog is comprised of one or more dialog boxes and a predefined dialog flow.

A dialog box (also known as a screen mask, or simply mask) combines numerous input and output elements. Each mask will usually correspond to one interaction step relating to a specialist function supported by the system and is usually implemented in the form of a screen page. The structure and content of each dialog box must be defined in the GUI specification. As well as the type, size, and position of dialog elements, it may also be necessary to include rules on their activation and deactivation. Consistent representation of the (technical) content is also important: Identical domain objects in different dialog boxes should have identical modeling wherever possible.

Data stored in the system database must be converted into a format that can be seen and read by users before it appears in a dialog box. For example, dates are often stored as the number of milliseconds that have elapsed since 1 January 1970, e.g. 8 May 2014; 10:08:29 GMT+0200 would be stored as the number 1399536509704.

Where system users are required to process specialist information, technical and conceptual validation of their entries is required. Validation ensures that the data entered is conceptually correct. For example, an input box for a whole number should not contain any symbols other than the numbers 0-9. As well as technical validation of the correct data type, conceptual validation is also needed. For example, when booking a flight ticket, the departure date cannot be in the past.

Typically, an information system’s GUI is comprised of multiple dialog boxes which support the user when processing a task. For example, when booking a hotel room, a flight, and a train ticket, the system guides the user through multiple screens with different dialog boxes until all bookings are complete. The options for navigating between dialog boxes are known as a dialog flow.

Operational systems: Systems that selectively support business processes and activities in organizations. There is typically a major emphasis on the value-creating and supporting processes within organizations.

A dialog flow depicts the sequence of operations requiring the processing of multiple masks in succession. The dialog box which appears in the next step is often derived from the values entered by the user in the preceding masks. For example, if a train ticket booking does not require a seat reservation, then the dialog box for seat selection is skipped and the dialog jumps to the payment information. Specification of a dialog flow encompasses:

* + Every mask used in the workflow
  + The defined sequence in which masks appear
  + Manual navigation options between masks (back, next) and
  + Dialog flow conditions to control when certain masks can be skipped.

Together with the conversions outlined above and the technical and conceptual validations, these elements make up a complete GUI specification. The documentation format and level of detail depend on the specific project situation and associated requirements.

###### Documentation formats for GUI specifications

User interfaces are specified both visually and in text form. The visual specification defines the appearance of the elements used and how they are arranged on the screen and may include sketches and screenshots of GUI prototypes and simple diagrams indicating the sequence of dialog boxes. The text specification explains correlations that cannot be expressed visually or whose visualization is too complicated. This includes rules on the activation and deactivation of GUI elements, validation rules, and dialog flow conditions. A simple example of a detailed GUI prototype can be seen in Figure 4.

Specification of user interfaces (GUIs)

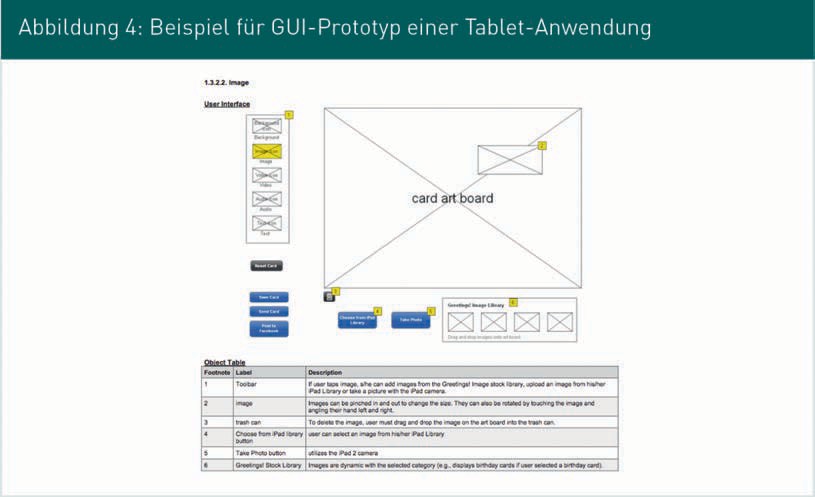
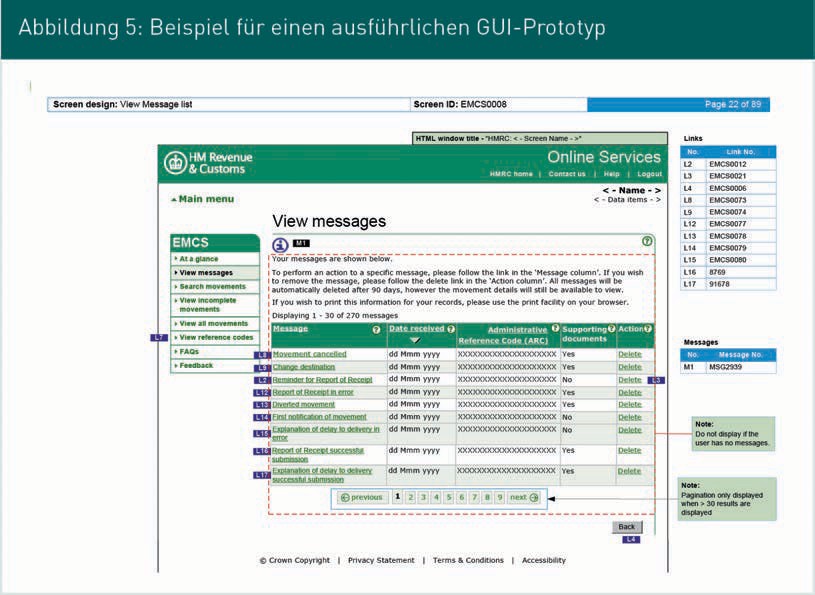


Figure 5 shows an excerpt from a very detailed GUI prototype.



Top tip:

A very detailed example of a GUI prototype can be found online (GOV, n.d.).

When using GUI prototypes, it is important to include a text description with the graphical representation, which should contain the following aspects:

* + Intended purpose of the GUI or reference to the business process
  + Required / depicted domain objects
  + Runtime parameters for activating/deactivating elements of the GUI
  + Dependencies on other GUIs and
  + Validation rules for technical and conceptual validation.

### GUI elements in individual dialog boxes

A dialog box as part of a dialog comprises various GUI components. There are three basic categories of GUI components:

* + Atomic components
  + Composite components and
  + Complex components.

###### Atomic GUI components

Atomic GUI components are simple elements for displaying and processing individual values. They cannot be broken down any further. Table 1 lists some typical atomic GUI components, while Figure 6 illustrates the interaction between these components in a simple GUI.

Specification of user interfaces (GUIs)



|  |  |  |
| --- | --- | --- |
| Table 1: List of key atomic GUI components | | |
| Title | Description | Example |
| Label | * Simple, non-editable text output * Suitable for descriptive texts, labeling of input boxes, display of read-only values | Applicant: |
| Text box | * One-line text entry * Suitable for all types of entries |  |
| Checkbox | * Display Boolean values * Change the status by clicking * Suitable for yes/no selections |  |
| Dropdown box | * Individual options for selection * Click to display the list of options then click again to select * Suitable for selecting mutually exclusive options |  |

|  |  |  |
| --- | --- | --- |
| Title | Description | Example |
| Multi-line text box | * Enter text with line breaks * Used for longer entries in natural language, such as comments |  |
| Button | * Button for activating functions |  |
| Link | * Text output with a linked interaction * Suitable for references |  |
| Image, icon | * Display graphical elements * Often also used as a graphical link * Suitable for references | (Source: Wikipedia) |

###### Composite components

Composite components: GUI components

comprised of multiple, conceptually connected atomic components.

Composite components are GUI components comprised of multiple atomic GUI components. They are used to structure the user interface and aid clarity. Table 2 contains some examples of typical composite components (option boxes, tables, and grouping).

Specification of user interfaces (GUIs)

|  |  |  |
| --- | --- | --- |
| Table 2: Examples of composite components | | |
| Title | Description | Example |
| Option boxes | * Display individual selection options * Click to switch to the selected option * Suitable where there is a choice of mutually exclusive options | (Source: Deutsche Bahn) |
| Table | * Display data in tabular format * Tables may contain input elements, output elements, or a mixture of both * Suitable for tabular data and for structuring the interface | (Source: Deutsche Bahn) |
| Grouping | * Multiple atomic GUI components are combined into a conceptual group * Suitable for visual structuring of the GUI * Ensures the uniformity of frequent identical inputs, e.g.   addresses | (Source: Deutsche Bahn) |

###### Complex GUI components

Complex GUI components are used to display and process complex values and data structures. They often implement a complex modeling and validation logic, such as a graphical calendar component for entering dates or a map component for entering a geographical location.

Table 3 lists some examples of complex GUI components for entering dates, navigation, and the editing of formatted texts.

|  |  |  |
| --- | --- | --- |
| Table 3: Examples of a complex GUI component | | |
| Title | Description | Example |
| Calendar | * Intuitive input of date values which ensures that only valid data is entered * Suitable for the input of dates | (Source: RichFaces) |
| Tree menu | * Intuitive navigation through complex data structures * Suitable for navigating through data records or selectively accessing specific dialogs | (Source: RichFaces) |
| Editor | * Complex component for entering formatted text including images and links * Suitable for editors of websites and other electronic documents | (Source: RichFaces) |

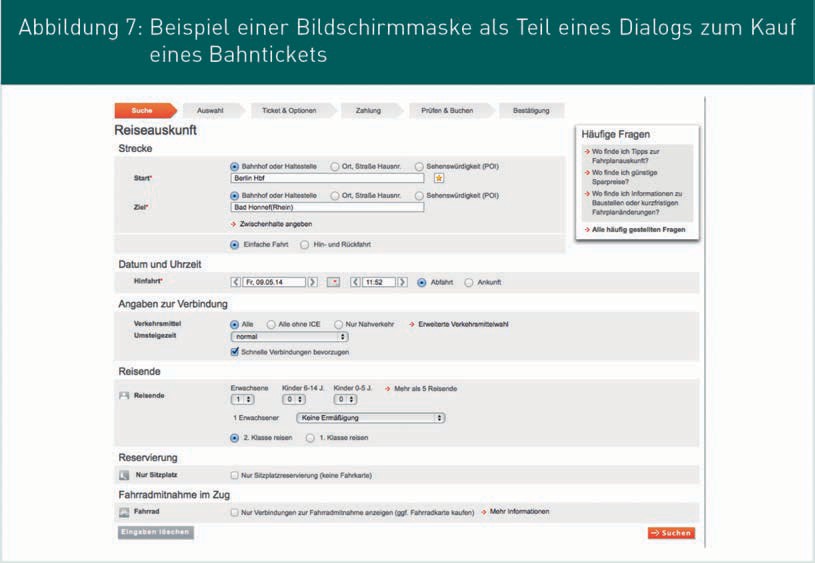
As well as the atomic GUI components provided by the majority of technological platforms, today’s GUI frameworks also offer a wide range of composite and complex components. It is therefore advisable to obtain an overview of the actual GUI components available before undertaking a detailed specification.

Specification of user interfaces (GUIs)

Top tip:

Comprehensive documentation of GUI technology is readily available on the Internet, including detailed explanations of the individual GUI components. Examples include:

* Java Server Faces (n.d.)
* RichFaces (n.d.)
* Microsoft User Interface Element Reference (n.d.)
* Apple Mac OSX UI Element Guidelines (n.d.)



###### GUI components for the input and output of simple data types

The specification should provide the development team with detailed requirements for the system’s implementation. Conceptual data models are often used as the basis when specifying GUIs. The GUI specification must enable users to enter every required value in the required format, ideally without errors, in the information system. Table 4 below summarizes the most common data types for simple values and the GUI components that are suitable for these inputs.

|  |  |  |
| --- | --- | --- |
| Table 4: Data types and input elements | | |
| Data type | Description | Data input element |
| String;  Up to 1000 characters | Short strings of characters | Text box Multi-line text box |
| Text (string);  1000+ characters | Long strings of characters | Multi-line text box |
| Whole numbers (integer, long) | Values representing whole numbers; the only permitted characters are 0-9 and signs (where applicable) | Text box |
| Numbers  (float, double) | Values representing numbers with a so-called floating decimal point; the only permitted characters are 0-9, “.” and signs (where applicable) | Text box |
| Monetary amounts | Monetary amounts, typically with two places after the decimal point | Text box |
| Date | Date values, typically given as day, month, and year; plus the precise time, where applicable | Text box  Calendar |
| Logical value  (3-state Boolean) | Logical values (yes/no); the system also records whether this value was consciously entered by the user | Checkbox (yes/no only)  Dropdown list (for 3-state Boolean)  Option button |
| 1-of-n selection; enumeration types (strings or numbers) | Select precisely one value from a list of predefined values | Dropdown list Option button Multiple checkboxes |
| m-of-n selection; enumeration types (strings or numbers) | Select multiple values from a list of predefined values. Note: Often implemented with multiple individual selections. | Multiple checkboxes |

Specification of user interfaces (GUIs)

As well as specifying GUI components for the entry of simple values, it is often necessary to specify how the relevant read-only view is displayed. A read-only view displays the entered data but cannot be edited. This type of view is often found at the end of an ordering process, for example in online shops, allowing users to review their order before finalizing it. While read-only views can usually be set for any input element, they are not always user-friendly. Text boxes are often shown against a grey background to indicate that they cannot be edited, but this makes them harder to read. Consequently, it is often necessary to define a suitable read-only view for each input element depending on the application. Table 5 provides a suggestion of how to denote the aforementioned input elements in a read-only view.

|  |  |
| --- | --- |
| Table 5: Suggested read-only view for input elements | |
| Data input element | Data output elements |
| Text box | Simple text, automatically scaled down, empty text represented as “---” |
| Multi-line text box | Simple text, automatically scaled down, allowing for line breaks  Empty text represented as “---”. |
| Calendar, date input | Simple text, formatted, date format typically depends on the user’s localization  Empty date represented as “---”. |
| Dropdown list | Simple text  No selection represented as “---” |
| Option button, checkbox | Pre-defined labels or icons for each selected option |

###### Specifying enumerations

The 1-of-n selection with predefined selection options is a common type of input element known as enumeration (see module 5.2). Unlike simple value inputs, when specifying GUI elements for the input and output of enumerations, additional aspects must be taken into account. In the following example, Table 6 highlights several key aspects which are relevant for the specification of enumerations.

Example

An application requires the user to select a specific value from a predefined list, such as the departure airport when booking a flight. The GUI should be made available in all European languages.

|  |  |
| --- | --- |
| Table 6: Aspects when specifying enumerations | |
| Aspect | Description |
| Conceptual label (per value)  Where applicable, allow for internation-alization | Text is displayed to the user in the GUI as a selectable value, generally depending on the chosen language. In this example: Name of the city in the chosen language.  Every selectable value must be translated into each of the languages supported by the application. Where relevant, a default setting should be defined if a specific translation is not available.  Example: Translate the city name into all relevant languages |
| Technical label (per value) | String used internally in the GUI to identify a selectable value; independent from the user’s language settings  In this example: International abbreviation for the city |
| Default setting  (per input element) | Where necessary, a default setting may be specified for each input element, and displayed to the user in the form of a preselection.  In this example: The closest airport to the user’s current location |
| Sequence of potential values (per input element) | The sequence of selection options must be specified for each input element. In some instances, the sequence may depend on the current GUI language. In our example: Alphabetical, ascending order in the chosen language |

When writing the specification, take care to consider all relevant requirements and settings for the input and output of enumerations. Particularly when lists of options are generated automatically or configured by administrators, the information required is fairly complex and should not be underestimated.

Specification of user interfaces (GUIs)

For multi-lingual, multi-cultural applications in particular, allowance must be made for various conceptual labels and consideration must be given to how this may influence the sequence of potential options.

###### Conversion

As well as inputting and outputting domain objects via the GUI, with simple values a very specific form of rendering is often required. If this differs from the format stored in the database, it will need to be converted for data input and output purposes.

Example

A simple German car number plate is comprised of the regional prefix code and the license number. The prescribed regional prefix code is comprised of 1-3 letters, while the license number is comprised of 1-2 letters and 1-4 numbers. The total number characters must not exceed 8.



When devising an expedient input option for a car number plate in a graphical user interface, it is first necessary to select a suitable presentation format, then define how the format stored in the data model is to be converted to that format. Table 7 illustrates various input options for German car number plates. The precise details of conversion will depend on how the display is implemented.

|  |  |
| --- | --- |
| Table 7: Options for entering a car number plate in a GUI | |
| GUI variant | Specification |
|  | One text box: “Regional prefix code” “License number” |

|  |  |
| --- | --- |
| GUI variant | Specification |
|  | Three text boxes:   1. Text box: Regional prefix code 2. Text box: Initial letters of license number 3. Text box: Digits of license number |
|  | Two text boxes:   1. Text box: Regional prefix code 2. Text box: License number |
|  | One dropdown list (left): List of all available prefix codes   1. Text box (center): Initial letters of license number 2. Text box (right): Digits of identifying number |

Other examples of application-specific conversion include reference numbers, customer numbers, insurance policy numbers, and order numbers.

The representation of date / time entries and monetary amounts will likewise require special conversion guidelines. These are often provided by special components such as the calendar component, as part of GUI frameworks.

### Validations

Certain GUI input elements, such as text boxes, allow users to enter any given values. These values must then be checked before they are transferred into the system and saved. Once the input elements of a dialog box have been specified, the validation rules for the dialog flow must also be specified. The system uses these rules as data is being entered by the user to gauge whether it meets the required quality from both a technical and a conceptual perspective.

Specification of user interfaces (GUIs)

As well as the actual validation rule itself, it is also necessary to specify the timing of evaluation, the consequences of violating the rule, and how users are told that a rule has been violated.

###### Constraint (condition)

A constraint (validation rule) refers to the actual rule or condition evaluated by the system. A constraint should facilitate a clear “true or false” evaluation and is therefore often formulated as a Boolean expression. We distinguish between different types of constraints depending on the type of GUI element and the conceptual requirements applicable to the dialog. The validation types are as follows:

* Required field check: Checks whether all the required values have been entered
* Conversion check: Checks whether the inputs produce valid values and
* Plausibility check: Checks whether the converted values are plausible (“make sense”).

Required field check

The required field check means checking whether the user has actually entered a value in the GUI element. In many applications, certain pieces of information are required by the system. Specifying GUI elements as required fields forces the user to make an entry, otherwise this step cannot be completed.

Examples: “Check whether the user has accepted the T&Cs”, “Check whether the name and email address have been entered.”

Conversion check:

The conversion check reviews the user’s entry for correct formatting. A successful conversion check must be completed before the entered value can be converted into the conceptual model, processed, and saved. The conversion check is particularly important for entries made in text boxes.

Examples: “Field content is a whole number”, “Length of input < 8 characters?”, “Field content is a valid email address.”

Plausibility check

The plausibility check relates to one or more input boxes and checks to ensure that the entries in individual fields do not violate conceptual constraints when combined. Conceptual plausibility checks are particularly important in the case of longer dialogs across multiple screens involving complex correlations.

Boolean expression A mathematical formula which may be evaluated as TRUE or FALSE. Example: (Price>2000) AND

(Customer status =

“Premium customer”).

Example of rules for checking a field: “Start date is a date in the future”,

“Value <=250”, “Date of birth at least 18 years ago”.

Example of rules for checking the correlations between multiple fields: “Contract start date precedes contract end date”, “(contract start date is before 1/1/2008) AND (contract start date is before contract end date)”, “Sum total (field1 + field2 + field 3) <= 100”, “For bicycles, the value “ICE” is not included in the set of potential modes of transport.”

###### Timing of the evaluation (transaction level)

The transaction level is determined by specifying the timing of the user interaction when the constraint is actually evaluated. Not all constraints must be met at all times. Plausibility checks in particular often require multiple value inputs which may even be entered in different masks. Typical timings (transaction level) include:

* + Exit the GUI component: Suitable for conversion checks
  + Exit the screen page: Suitable for required field validation, conversion checks, and plausibility checks
  + Data buffering, e.g. to resume processing at a later date: Conversion checks
  + Submit data to the system and exit the input process: Required field validation, conversion checks, and plausibility checks.

The specific timing of validation typically depends on both the conceptual requirements and the GUI technology used. Entries cannot always be validated immediately upon exiting the GUI component and may require a manual page change by the user.

###### Validation type (error or message)

As well as forcing the input of certain values by the user, validation rules can also be used to display automatic messages when fields are completed, because not every rule violation constitutes an error or prevents processing of the entered data. In certain cases, for example, users could be alerted to the possibility of delayed processing or the requirement to retrospectively submit certain records by post or email. When booking rail tickets or hotel rooms, a typical example would be a message indicating that capacity levels are high and there are only a few rooms or seats still available.

Specification of user interfaces (GUIs)

Each validation rule may therefore be specified as either an error or a message. While recognized errors must be rectified before ongoing work can continue, messages do not affect data storage and transmission.

###### Displaying failed validations

Where a constraints violation has been identified, it must be conveyed to the user in a suitable format depending on the validation category and type (error, message). While an error message is often sufficient for required field validations and conversion checks, plausibility checks usually require a more detailed description.

The specific text and its position, font, color, and size must also be taken into account when specifying validation rules. Additionally, it is necessary to specify how the error message and affected elements are displayed to the user. A popular approach is to draw a box around the affected input fields and highlight the input elements in a different color. Some examples can be seen in Figures 9, 10, and 11.





Task

Practise specifying a simple login dialog that allows registered users to log in with a combination of their username and password. Then present your results in the tutorial.

### Navigating between dialog boxes

The operations in an application often comprise multiple steps which must be executed in sequence. For example, when purchasing a rail ticket, the first step is to request basic information about the journey, then select from the available connections, then enter any seat reservation requests, followed by the payment details, and finally the online ticket is generated. For clarity, this process can be broken down into a series of pages so that users are not confronted with too many complex boxes at once. It also allows support to be provided for each individual step with more detailed prompts. With each page change, the data already entered is processed and the GUI elements on subsequent pages are updated to reflect the current data model. For example, when booking a rail ticket, the second step will display specific data and information based on the real connections available.

Specification of user interfaces (GUIs)

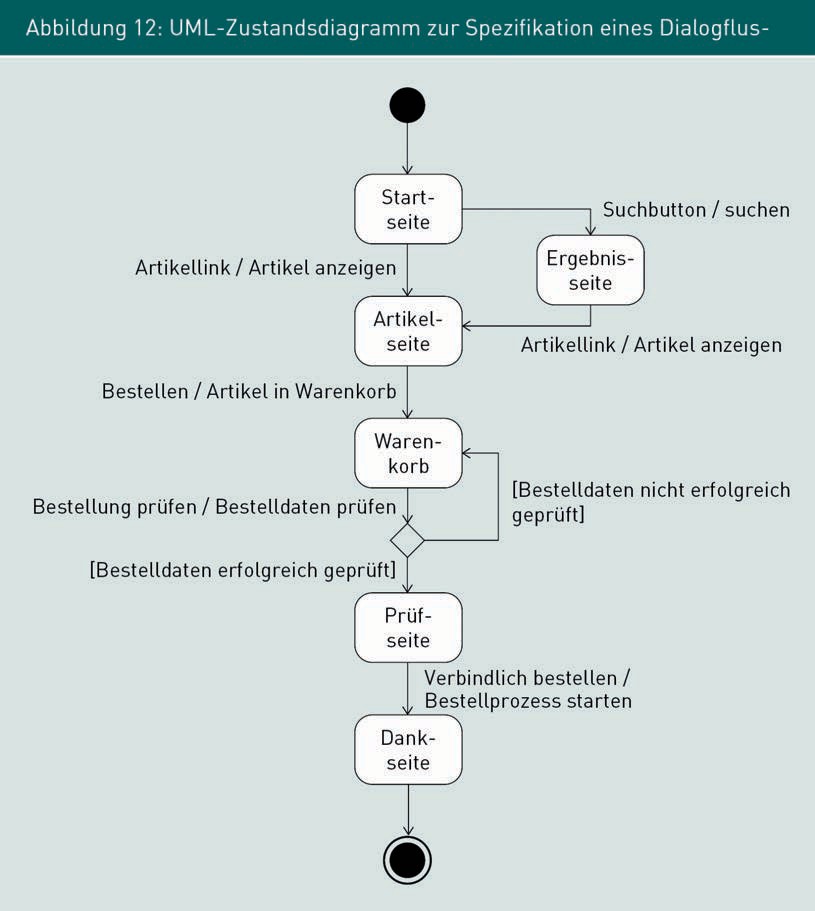
###### Dialog flow

A dialog is sub-divided into multiple dialog boxes. The specific sequence and options for navigating between individual dialog boxes are determined by the dialog flow. The following aspects must be taken into account when specifying a dialog flow:

* Each box used in the workflow
* Specification of the sequence in which boxes are displayed
* Manual navigation options between boxes (back, next) and
* Dialog flow conditions to control the flow (skipping dialog boxes).

###### Specifying dialog flows in UML state diagrams

The aim when specifying dialog flows is to obtain a clear, precise, human-readable description of all potential operations through the dialog and the relevant dialog flow conditions. UML state diagrams offer a simple option for the structured, visual specification of dialog flows. Figure 12 illustrates an example of a simplified dialog flow for an online shop. Each dialog box is represented by a state, while the navigation options between individual dialog boxes are modeled by a transition. A transition between two dialog boxes means that users can move from one box to the next in the direction indicated by the arrow. Transitions are triggered when navigation is activated, generally as a result of the user clicking on a button or link. Where necessary, navigation can also be activated by the system without any user input, for example when a certain period of time has elapsed. When transitioning from one box to the next, it is possible to model the requirement to activate a technical function or perform certain checks in the form of an activity. For example, in Figure 12, users click on an item link to transition from the results page to the item page. This activates the “Show item” function and full details of the selected item are then displayed. Table 8 summarizes the principal components of a state diagram and how they are used in the specification of dialog flows.



|  |  |  |
| --- | --- | --- |
| Table 8: Components of a UML state diagram for specifying a dialog flow | | |
| Name | Meaning | Icon |
| Dialog box | One state corresponds to one screen mask. |  |

Specification of user interfaces (GUIs)

|  |  |  |
| --- | --- | --- |
| Name | Meaning | Icon |
| Initial status | Indicates a potential entry page |  |
| Final status | Indicates that the dialog is conceptually complete. Dialog flows can usually be interrupted at any time, e.g. by closing the application. |  |
| Navigation | A transition specifies the option for navigating between two dialog boxes. |  |
| Trigger [condition] / Activated function | * The trigger activates navigation, typically by clicking on a button or link. * A guard is a constraint that must be true in order to execute navigation. * The activated function is a function of the business logic which is triggered upon exiting a dialog box. When the function is complete, the target box is displayed. | Item link / Show item |
| Decision | A decision allows users to choose the next dialog box depending on the current result or current state of a business object. |  |
| Merge | Merge enables the amalgamation of multiple navigations provided they each have an identical target box |  |

###### Challenges when specifying dialog flows

A typical challenge we face when specifying dialog flows is how to describe standard and exception navigations. A typical standard navigation might access the help pages or the legal notice of all other dialog boxes. Exception navigations arise when the dialog is aborted because the application is closed, or due to a timeout or long periods of inactivity by the user or the occurrence of unscheduled errors, such as failure to access the booking system when making a hotel reservation.

As these types of navigation could occur on any page, a state diagram can soon become extremely complex if we attempt to model all standard and exception navigations. Alternatively, commonly occurring navigations and those shared by all dialog boxes can be modeled as examples of one dialog box and then generalized for all other boxes in the descriptive text. As with requirements engineering, the main emphasis is on interpersonal communication. When modeling dialog flows, therefore, it is important to concentrate on a relevant rendering appropriate to the current communication situation.

Summary

An application’s GUI is usually comprised of multiple dialogs, each made up of one or more dialog boxes and a predefined dialog flow. A dialog box includes numerous atomic GUI elements, composite elements, and complex GUI elements, usually on one page. As well as defining rules for converting data stored in the system database into a format that can be displayed and read by the user, it is also necessary to specify validation rules. A distinction is made between required field checks, conversion checks and plausibility checks. The options for navigating between dialog boxes are specified as a dialog flow; the UML state diagram is one option for a structured documentation format.



# Unit 3

## Specification of system components

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... what we mean by “component” and the correlations between systems and components

... how to specify component structures using the UML component diagram

... which documentation formats can be used to specify the internal behavior of components

DL-D-ISPE01-L03

1. Specification of system components

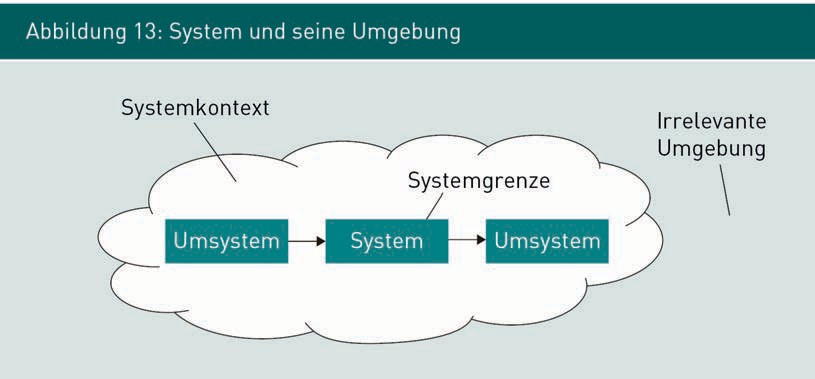
### Introduction

Because industrial software systems are so complex, they are divided into different parts, or components, each of which can be developed, maintained, and operated independently. This unit begins by exploring precisely what components are and then goes on to examine the structure and internal behavior of components.

### Introduction to components and motivation

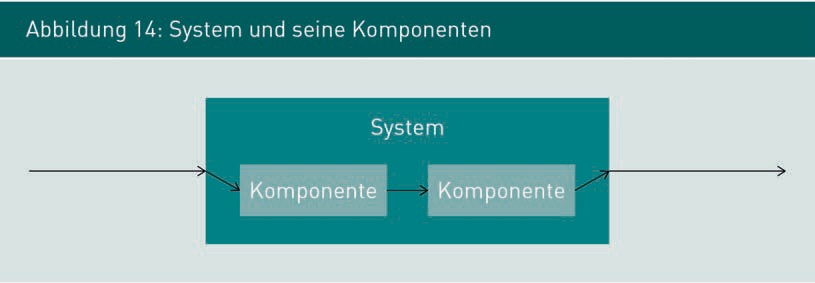
###### System versus component

A system is an interactive relationship which is delimited from its environment and is in turn comprised of other interactive relationships. Figure 13 shows a simplified illustration of a system within its system context.



A specification need not necessarily describe the planned software system in its entirety. Only certain parts of the system may be relevant, depending on the individual project. With this in mind, complex software systems are broken down into so-called components. Each component is an independent software unit which can be combined with other components via agreed interfaces to create a software system. Figure 14 illustrates the relationship between components and their related system. In an object-oriented system, classes are the smallest system components. Typically, multiple classes which interact to perform very specific functions are combined to create components.

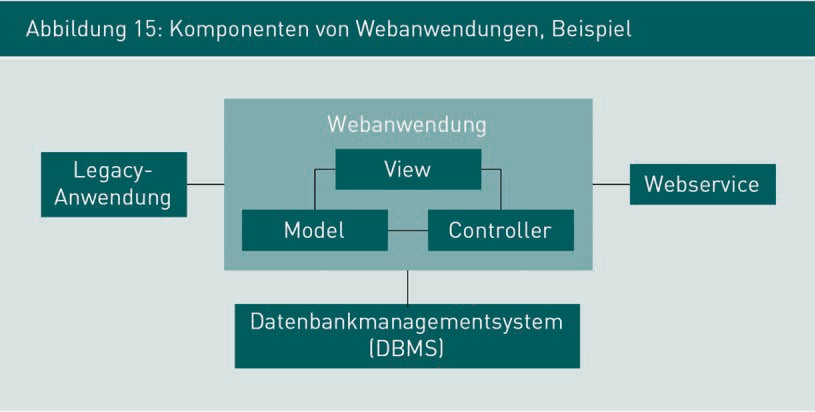
Specification of system components



Components

Independent software units which can be combined with other components via agreed interfaces to create a software system.

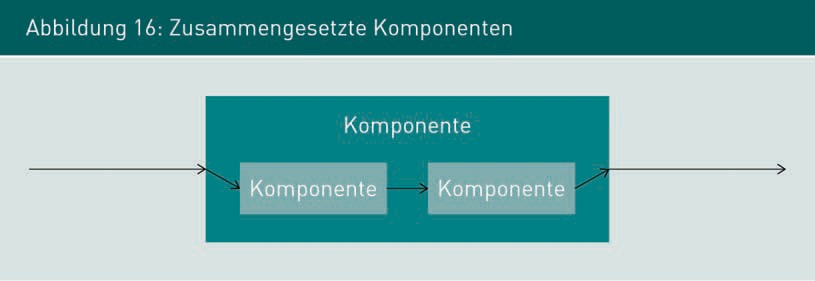
Figure 15 denotes the interactions between four different systems: a legacy application, a web application, a database management system, and a Web service. The Web app in turn is comprised of three components: Model, View, Controller.



Conceptual requirements engineering defines the boundary between the system as a whole and its environment, i.e. the system context. It also determines which elements must be included in the specification, in other words, where the specification “begins” and “ends”. A component is the smallest system unit that can be specified. SRS specifies externally visible behavior and structures, while the component’s internal structure remains hidden. The software architect and/or development team decides on the component’s internal structure and implementation of the required functions.

###### Component properties

A component is a part of a modular system whose content remains hidden from the user. A component may in turn be made up of other components. Figure 16 depicts a composite component comprised of other components. A component’s smallest element is known as a class.

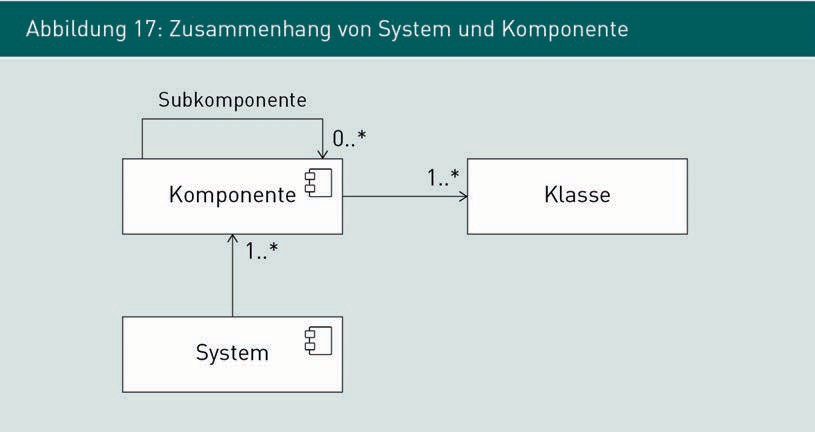


Like classes, components deliver functions via an interface. While the functionality of classes is achieved via their methods, components are addressed via interfaces. A component’s interface description clearly defines its external functionality. One component may be replaced by another, provided it implements the same interfaces.

Object orientation is a way of viewing complex systems as an interaction between cooperating objects. Within the context of industrial software systems, this perspective can also be generalized for components. One characteristic of industrial information systems is that they break internal elements down into zones (components), each of which is responsible for one very specific function, and the correlations between individual components.

Figure 17 illustrates these correlations in the form of a class diagram. A system must comprise at least one component, while a component may contain various sub-components. Classes are the smallest possible structural elements in an object-oriented system.

Specification of system components



Larger components often operate as independent applications. For example, data is often stored in database management systems (DBMS), which are independent applications used solely by other applications. Figure 15 illustrates one such scenario. The Web app communicates with the DBMS via a clearly defined interface. In principle, the DBMS may be exchanged without affecting the functions of the Web application.

The consistent use of components is a comparatively easy way of achieving the quality goals of a software system that is

* Readily extendible
* Easier to test and
* Easier to maintain.

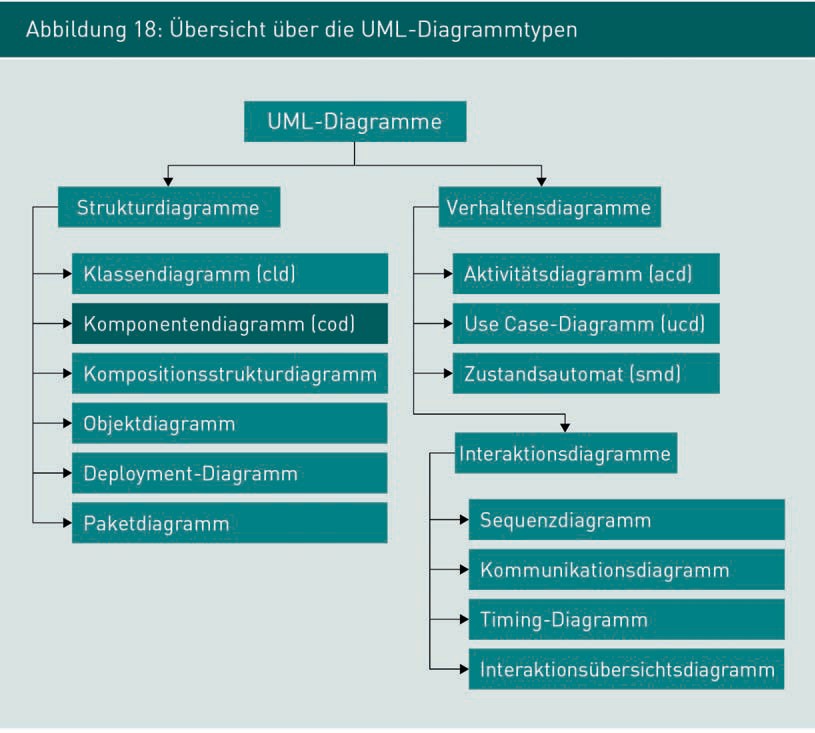
Because industrial information systems are so complex, they can usually only be controlled by breaking them down into individual components.

###### Specification of components

Specification identifies and delimits conceptual components from other conceptual components (examples: shopping basket, item, customer). Depending on the nature of the project, technical components may also be specified, particularly if the internal system architecture is already known, or if it is only necessary to specify parts of the system rather than the whole system in a project. The first step is to specify the component’s internal behavior — for example, extend the term of the contract or calculate the total. The second step must clearly define the operations and logical structure of messages exchanged at component interfaces.

### Specifying the structure of systems and components

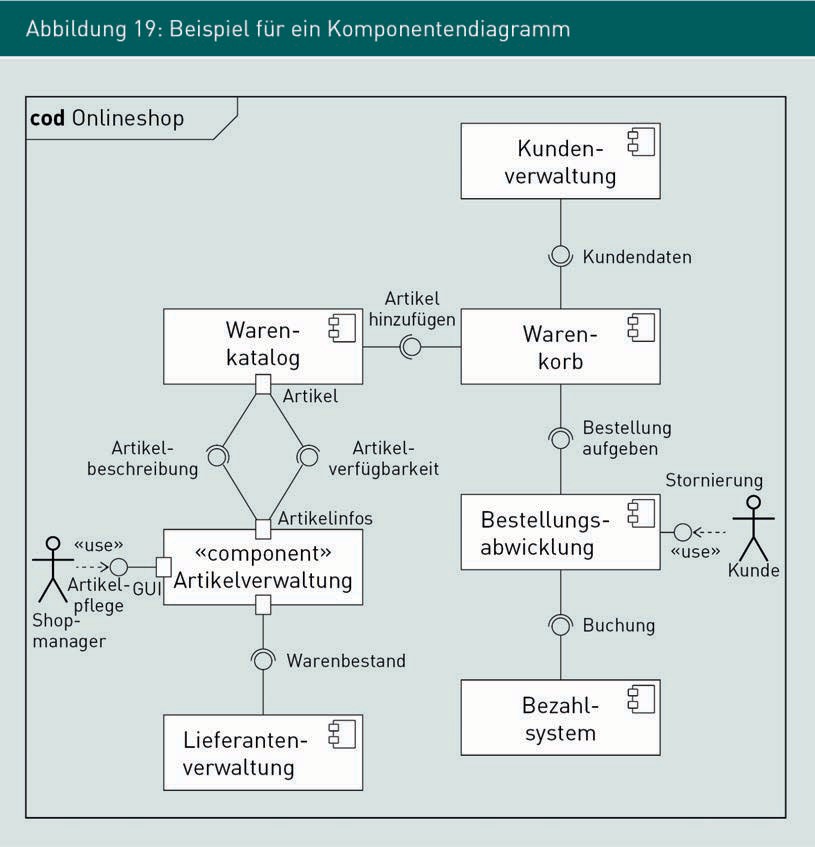
A UML diagram can be used to specify the components in a system, their sub-components, and the interface relationships to one another. The UML component diagram (cod) is a structural diagram; see Figure 18.



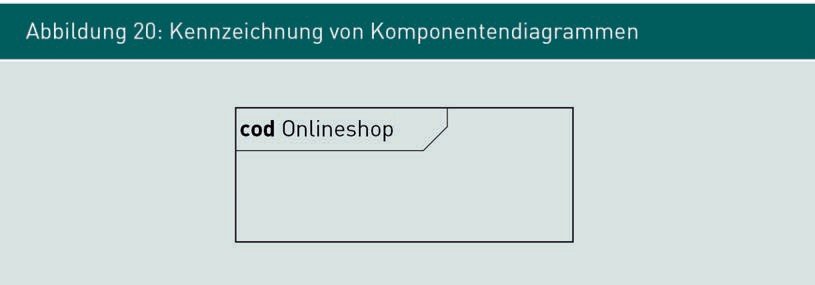
###### UML component diagrams: Model elements and use

UML component diagrams have two main elements with different types of representation: Components and interfaces. Figure 19 shows an example of a UML component diagram for an online shop and the relationships between the different components. The key elements for this type of diagram are shown below.

Specification of system components



In order to define a UML diagram and distinguish it from other diagrams, a frame is drawn around the component diagram, and the diagram type (cod) and the name of the diagram are written in the top-left corner. The diagram should be named after the system being refined in the component diagram. If a component is broken down into its constituent parts, the diagram should be labeled with the name of the component. Figure 20 illustrates use of the diagram frame.



Note

In the following section, we state that the framing of UML diagrams, indication of the diagram type and diagram labelling are mandatory for all UML diagrams. In industrial practice, all diagrams should be labelled in the same way.

There are various options for component notation. The main ones are listed and explained below in Table 9.

|  |  |
| --- | --- |
| Table 9: Rendering of component notations | |
| Rendering | Description |
|  | Box (cf. UML class diagram) with a small icon in the top right. The icon identifies this “class” as a component. As in the class diagram, the name of the component is written directly into the box. |
|  | Box with the name of the component. The label <<component>> identifies the box as a component. This type of label is equivalent to an icon.  Note: In UML, the notation <<name>> means “stereotype” and allows notation elements to be given a specific name in the diagram. Stereotypes can be used in all UML diagrams. |

Specification of system components

|  |  |
| --- | --- |
| Rendering | Description |
|  | The stereotype identifies this box as a sub-system. Readers of the diagram can see immediately that this component is part of another component. |

Apart from components, interfaces are the other main focus of the component diagram. As with components, there are various options for rendering interfaces. Table 10 illustrates the principal notation options for interfaces in a component diagram.

|  |  |
| --- | --- |
| Table 10: Rendering of interfaces in a component diagram | |
| Rendering | Description |
| Box with the name of the interface. The box is identified as an interface by the stereotype <<interface>>. The notation is the same as the class diagram. | This rendering indicates it is an interface named “Item availability”.  Note: Certain interface techniques are not usually modeled in a component diagram. |
| A component is connected to an interface by a dotted line and a closed arrowhead at the interface. An alternative rendering might be:    Here, the component is linked to the interface with a solid line and a small circle labeled with the name of the interface. | This rendering indicates that the “inventory management” component is making the “item availability” interface available. Other components can access the functions of the component via this interface.  This form of rendering is also known as lollipop notation and is equivalent to the previous representation with the dotted line and arrow. It is often used because it takes up less space. |

|  |  |
| --- | --- |
| Rendering | Description |
| Here, the component is linked to the interface by a dotted line with an open arrowhead at the interface. The arrow is labeled with the stereotype <<use>>. An alternative rendering is:    Here, the component is linked to the interface by a solid line with an open semi-circle labeled with the name of the interface. | This rendering indicates that the “Shopping cart” component accesses the functions offered by the “Item availability” interface. The functions of this interface are vital for correct functioning of the shopping cart. This rendering is equivalent to the one above: It indicates that the “Shopping cart” component needs the functions of the “Item availability” interface. |
| Two components are linked via a named interface. | This rendering indicates that the “Shopping cart” component accesses the “Inventory management” component via the “Item availability” interface.  Tip for remembering the notation symbols: Visualize the semi-circle as a hand “grasping” the interface. |
| A component has two ports, XML and GUI, which are modeled as boxes on the boundaries of the component. Individual interfaces are assigned to specific ports. | Ports are used to combine individual interfaces and represent the externally visible access to components. In this example, the interfaces are grouped according to type: technical interfaces and GUIs. Interfaces can also be grouped by function. |

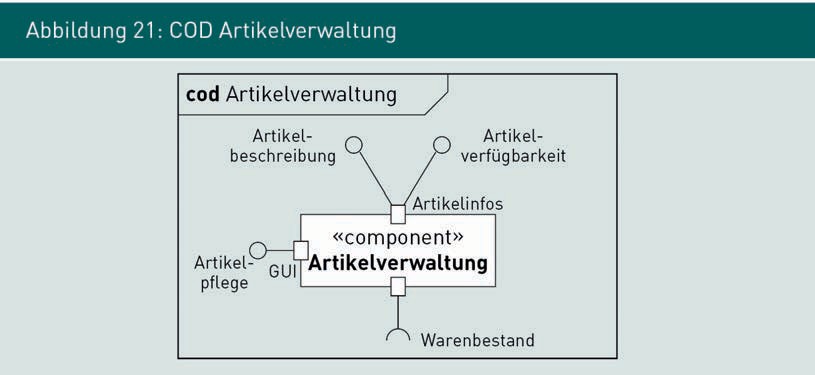
Specification of system components

###### Options for using the UML component diagram

Because the component diagram is a structural diagram, modeling focuses on the internal layout and elements of a system. When modeling functional components, their inter-relationships are conceptual dependencies which would exist even without the support of IT systems. For example, specifying technical components allows software architects to define the system’s actual components and their interfaces. Operations and actual data objects are not usually represented in component diagrams. The next module considers the options available for specifying component behavior.

### Specifying component behavior

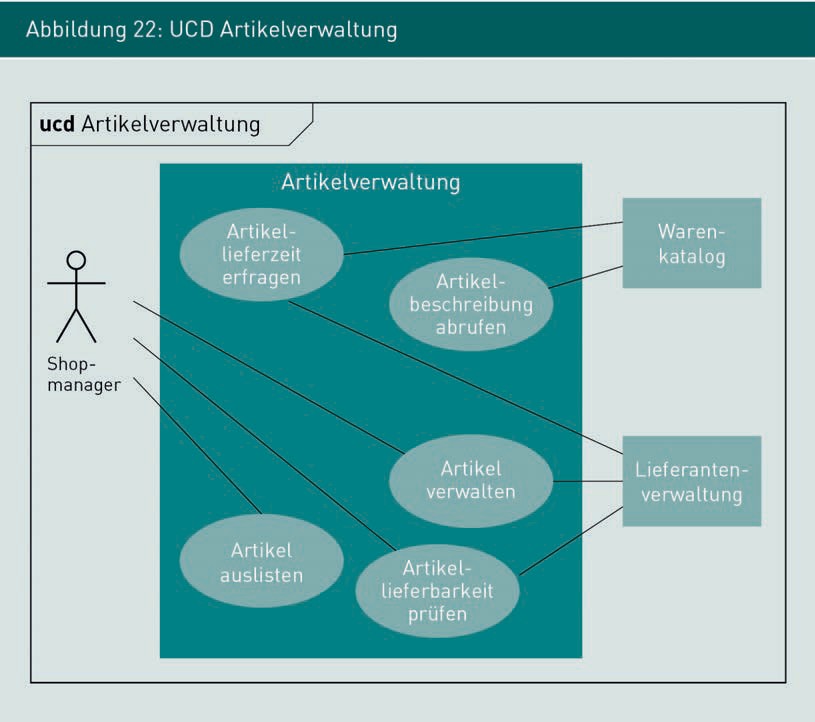
Below, we consider the options for specifying components using as our example the “Inventory management” component shown in Figure 21. Inventory management has three interfaces, one of which is explicitly labeled GUI. Additionally, the functions of the Inventory management interface are required by another component.



###### Using UML behavioral diagrams for modeling

Just as with complete systems, the behavior of an individual component can also be modeled in a UML use case diagram (see example in Figure 22). This diagram depicts the component overview and is modeled using the same notation elements as a complete system. Here, the system boundary represents the component boundary.

As with the use case representation of systems, use case diagrams for components indicate the required user interfaces as well as the technical system interfaces.



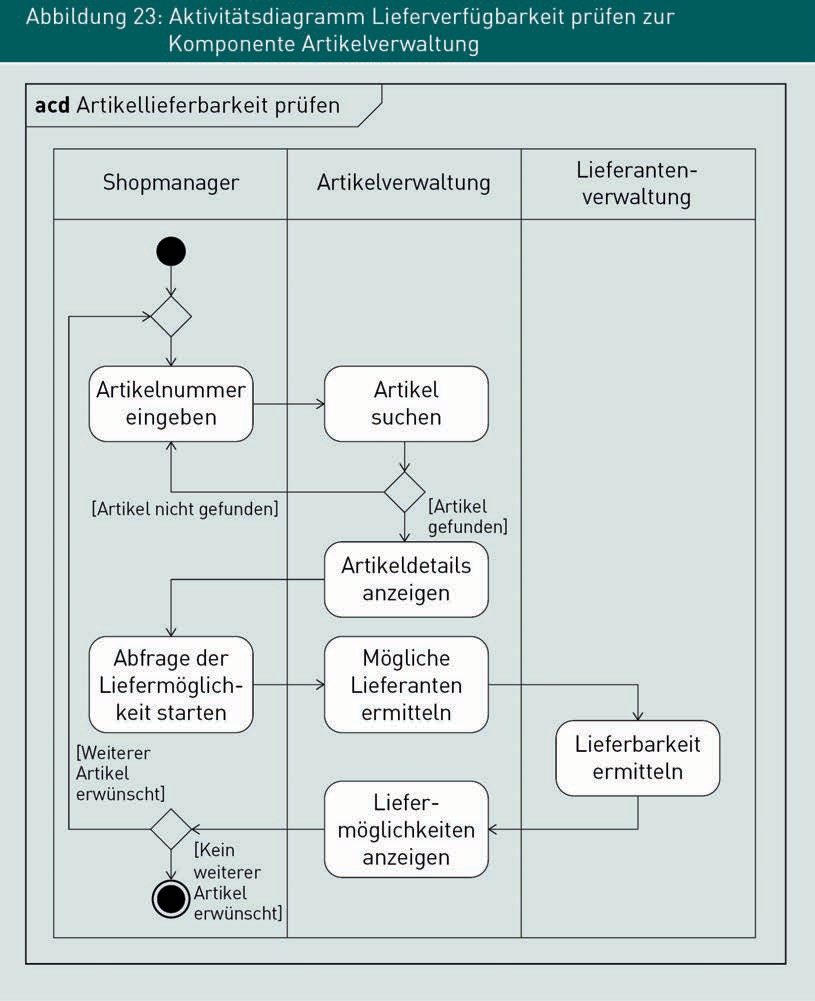
The activity diagram allows the component’s internal operations to be specified in greater detail, generally using the same notation elements as conceptual requirements engineering.

The activity diagram in Figure 23 illustrates a refinement of the use case

“Check item availability”. The diagram shown here contains three partitions

(“swimlanes”) as a new model element.

Specification system components



Partitions are used to specify and denote responsibility for functions and functional decision-making. Table 11 explains the use of partitions as notation elements.

|  |  |  |
| --- | --- | --- |
| Table 11: Partitions in an activity diagram | | |
| Name | Meaning | Rendering |
| Partition  (“swimlane”) | In an activity diagram, a partition delimits the influence of participating actors, which may include both human users and technical systems. |  |

Using partitions allows you to define precisely which actor is specifically responsible for which function. The specific requirements placed on interfaces are derived from these responsibilities. For example, Figure 23 shows us that the component “Supplier management” is needed to ascertain availability. On the other hand, the shop manager as a user of inventory management initiates a targeted inquiry into the availability of selected items. Decision-making responsibilities can also be modeled in the ACD with partitions. The above example entails two decisions: The first is made by the “Inventory management” component, and the second by the “Shop manager” actor.

Note

In activity diagrams, the UML also enables you to model business objects required for an action using so-called pins. Further information on the use of pins can be found in the UML standard and in the relevant literature.

###### Modeling with business rules

A business rule is a statement that defines or constrains a functional aspect. Business rules can be used to give assurances regarding the structure of business objects and influence the behavior of business processes. As well as structural requirements (e.g. “A contract must always be assigned to a policyholder”), business rules can also be used to specify operational requirements (e.g. “Damages in excess of €500,000 must always be assessed by two administrators” or “Contracts with a public-sector client can only be signed by the head office”). In practice, business rules are specified in text form.

Specifying system components

Operational business rules influence the control flow of business processes. In Figure 23, the “Inventory management” component procedure contains two decisions, at which point a rule is used to determine what happens next. As well as controlling operations, business rules are also used to control how business objects are handled. Business rules must have a context, one or more conditions, and one or more actions which are executed depending on the condition being met. Table 12 explains the elements of a business rule by way of an example.

|  |  |
| --- | --- |
| Table 12: Elements of a business rule for operational control | |
| Elements of a business rule | Example |
| Context:  Link to a function or situation. | “Customer management during order processing with long delivery times”: |
| Condition:  In which circumstances can changes be made to business objects? | IF  the client has the status “premium customer” AND the expected delivery time is more than 3 days, |
| Action/s:  Which changes must be implemented? | THEN  reduce the invoice total by 10% AND send a message via email OTHERWISE  only send a message via email about the longer delivery time. |

It is vital to ensure the completeness and consistency of business rules, particularly when business logic is responsible for evaluating business rules and implementing actions.

Business rule A statement which defines or constrains a conceptual aspect,

such as assurances on the structure of business objects or the behavior of business processes.

Completeness

In this context, completeness means that the rule must incorporate every possible combination of conditions and provide a corresponding action for each condition. With complex sets of rules, there may be very large numbers of possible combinations.

Example of an incomplete business rule

Digression

IF

the order value of the item is > €2000 AND the customer is aged > 18 years

THEN

offer the customer the additional option of payment in instalments.

IF

the order value of the item is <= €2000 THEN

offer the customer the option of payment in 8 weeks.

The business rule is incomplete because it does not specify the actions for an order value of > €2000 and a customer age of <= 18 years.

Therefore, when performing quality assurance on business rules, always explicitly check that every possible combination of conditions is covered by an action.

Consistency

Evaluating the set of rules at a given point in time must always produce precisely one outcome. In other words, precisely one action or group of actions must always be defined.

Example

Digression

IF

Specification of system components

the order value of the item is > = €2000 AND the customer is aged > 18 years

THEN

offer the customer the additional option of payment in instalments.

IF

the order value of the item is <= €2000 THEN

do not allow the option of instalment payments.

If the order value is precisely €2000, this business rule will produce inconsistencies.

For this reason, the decision in favor of one action particular action from the group of business rules must always be unequivocal. Exhaustive consistency checks can be very time-consuming, particularly if there are large numbers of rules. For this reason, it is important to carefully check the completeness and consistency of business rules as they are written.

###### Modeling with decision tables

Decision tables are another technique for specifying component behavior and documenting complex business rules. A decision table distinguishes four different aspects:

* Definition of conditions
* Definition of actions to be implemented
* Combination of results after evaluating conditions
* Allocation of evaluation results to specific actions.

Table 13 illustrates an example of a decision table setting out all the decisions for admissible payment options in an online shop. All relevant conditions which may influence the decision are shown in the top-left. All possible and/or relevant combinations of rule evaluations are shown in the top-right (in this example, all options which could potentially arise from the evaluation of three different conditions). The bottom-left of the decision table shows all relevant actions (in this example, the various payment options available depending on which conditions are met). Finally, the bottom-right specifies the decisions themselves, i.e. which payment options are offered for which combination of evaluation results.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 13: Example of a decision table | | | | | | | | |
| Conditions |  |  |  |  |  |  |  |  |
| Order value >= €2000 | N | Y | N | N | Y | Y | N | Y |
| Customer age >= 18 years | N | N | Y | N | Y | N | Y | Y |
| Customer status == Premium customer | N | N | N | Y | N | Y | Y | Y |
| Actions |  |  |  |  |  |  |  |  |
| Payment in installments |  |  |  |  | X |  |  | X |
| Payment in 8 weeks |  |  | X |  | X |  | X | X |
| Payment by invoice |  |  | X | X | X | X | X | X |
| Payment in advance | X | X | X | X | X | X | X | X |

Payment in advance is possible in all cases. Later payment is possible provided the customer is over 18 and payment in installments is only possible provided the order value is >= €2000 and the customer is over 18.

The decision table is useful for illustrating complex correlations between large numbers of business rules and checking them for completeness.

###### Modeling with state transition tables

State transition tables are another option for specifying component behavior. State transition tables can be used to specify potential actions or functions depending on the current state of business objects, components, or processes. A state transition table contains three different areas:

* + List of all potential states
  + List of all potential actions, and
  + Decision as to which actions are possible in which states.

Specifying system components

Table 14 shows an example of a state transition table to determine which actions the user may initiate depending on the status of an item in an online shop.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 14: State transition table for an item in an online shop | | | |
|  | Actions | | |
| States | Cancel | Order | Change delivery address |
| In basket | N | Y | Y |
| Being processed | Y | N | Y |
| Out for delivery | N | N | N |
| Paid | N | N | N |
| Canceled | N | Y | N |
| Returned | N | N | N |

This state transition table shows us that the action “Cancel” can only be applied to the state “Being processed”. The delivery address can only be changed in the states “In basket” and “Being processed”.

State transition tables can be derived directly from any UML state diagram. However, in our example in Table 14 a state diagram cannot be prepared, because the achieved target state has not yet been documented in the “admissible actions” column. Table 15 shows a similar state transition table where the relevant target states have been documented and could therefore be converted into a UML state diagram. The type of rendering chosen depends on the project’s current communication situation. A state diagram supports the graphical illustration of complex correlations, while a state transition table allows the user to see at a glance which actions are inadmissible in which states.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 15: State transition table for an item in an online shop | | | |
|  | Actions | | |
| States | Cancel | Order | Change delivery address |
| In basket | – | Being processed | In basket |
| Being processed | Canceled | – | Being processed |
| Out for delivery | – | – | – |
| Paid | – | – | – |
| Canceled | – | Being processed | – |
| Returned | – | – | – |

Summary

Complex software systems are broken down into components, whereby each component is an independent software unit which can be combined with other components via defined interfaces to create a software system. It is necessary to specify the internal behavior of each component, together with the operations and logical structure of exchanged messages at component interfaces.

A UML component diagram is one option for specifying the components of a system, their sub-components and their interface relationships to one another. UML behavioral diagrams, business rules, decision tables and state transition tables are all options for specifying the behavior of components.

Specification of system components



# Unit 4

## Specification of technical system interfaces

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... how to use UML behavioral diagrams to specify behavior at interfaces.

... what UML sequence diagrams are and how they are used in the specification of interfaces.

... how to specify data structures at interfaces with class diagrams.

DL-D-ISPE01-L03

1. Specification of technical system interfaces

### Introduction

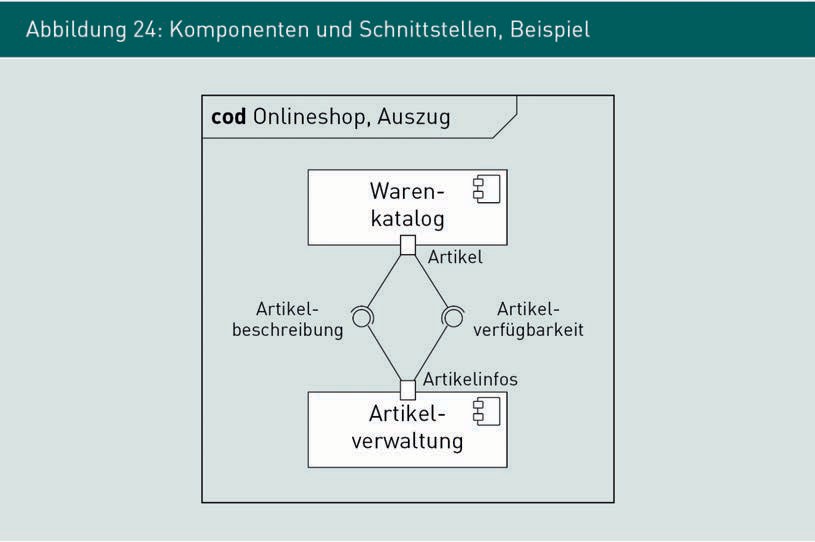
Up until now, we have focused on the specification of behavior and structures within systems and components. This unit explores the specification of behavior and data structures at technical interfaces. The approaches outlined here may be used both for interfaces between system components and for the interfaces between two systems. The term “interface” is therefore used as a generic term encompassing all technical interfaces.

### Specification of behavior at component interfaces

The specification of technical interfaces between components and systems (referred to collectively as “interfaces”) defines which messages can be exchanged between components (or systems) in which situations, as precisely as possible and in an appropriate format for the current communication situation.

The component diagram in Figure 24 illustrates the use of technical interfaces: In an online shop, the “Product catalog” component accesses the “Inventory management” component via two interfaces, but the development team needs more information about which data is exchanged via the interface and in which situations in order to implement the components.

Specification of technical system interfaces



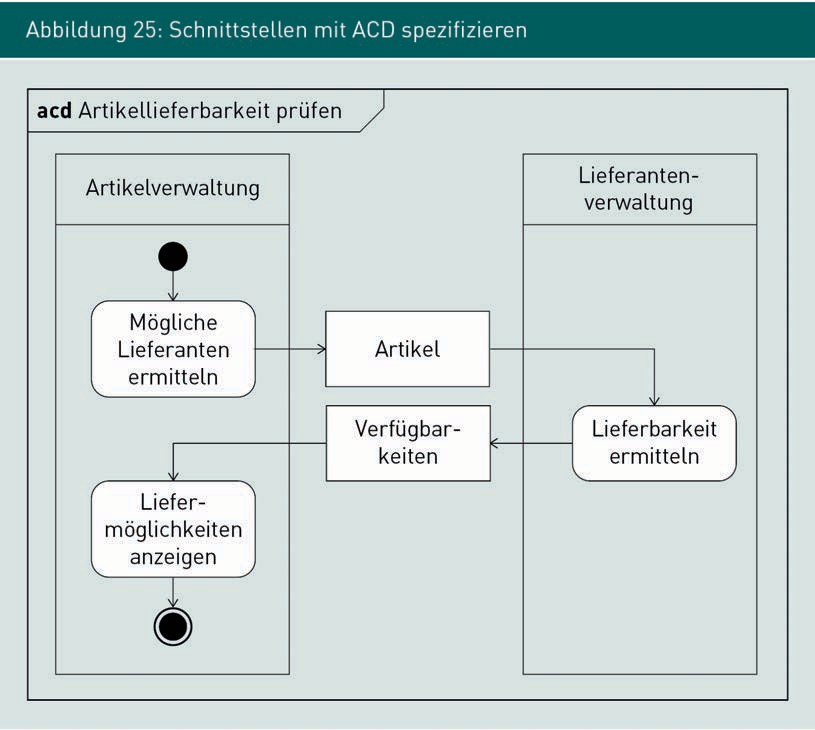
UML activity diagrams and UML sequence diagrams are suitable for the structured documentation of which messages are exchanged between systems and when.

###### Modeling with the activity diagram

UML activity diagrams are one option for specifying the exchange of messages at interfaces. An example is shown in Figure 25. Each component and its area of responsibility is depicted by a partition (“swimlane”). Each partition models the conceptual context within which messages between components are exchanged. In this particular example, communication between components is required when transitioning from the function “Identify potential suppliers” to the function “Determine availability.” Which messages (i.e. which data structure) are exchanged is defined in the ACD using object nodes. Table 16 explains the use of object nodes.

Object nodes

Notation elements in an activity diagram representing information that is required and output by actions.



Specification of technical system interfaces

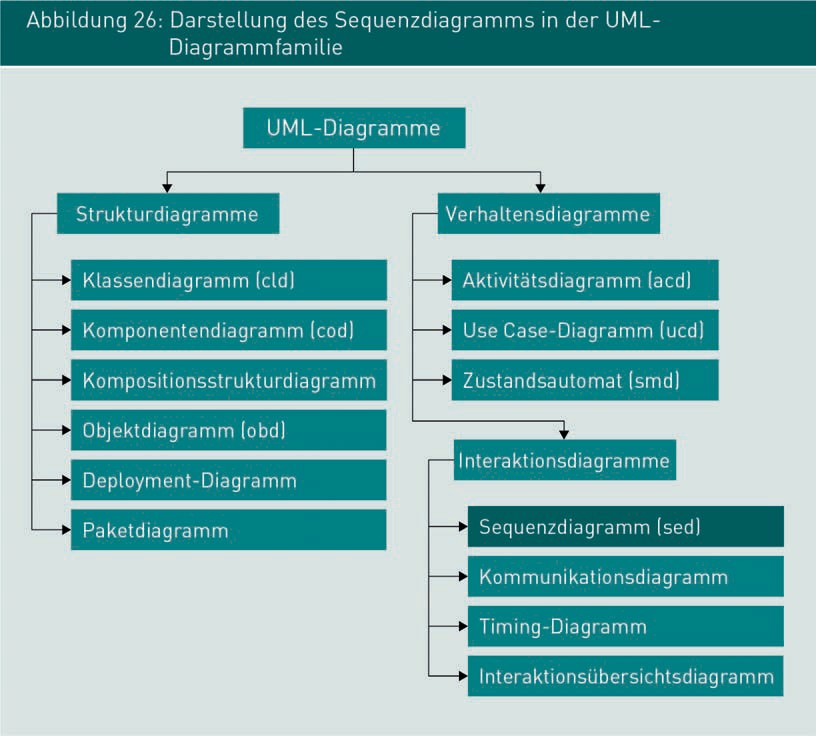
|  |  |  |
| --- | --- | --- |
| Table 16: Object nodes in ACDs | | |
| Name | Meaning | Rendering |
| Object node | In activity diagrams, an object node represents the type of information required by an activity before it can be executed and the type of information that will be output after execution. Object nodes are rendered by a box labeled with the name of the object type. |  |
| In the example on the right, the action Check availability requires an object of the type Item and outputs an object of the type Availability. |

Using partitions and object nodes in the activity diagram is a simple way of modeling complex operations whilst including the messages exchanged between systems. This creates a simple overview of the required message types. ACDs are less suitable for denoting the technical details of a specific message exchange or details about the content of individual messages. While theoretically possible using the notation elements in UML, the UML sequence diagram is a more suitable form of documentation for this purpose.

###### Modeling with sequence diagrams

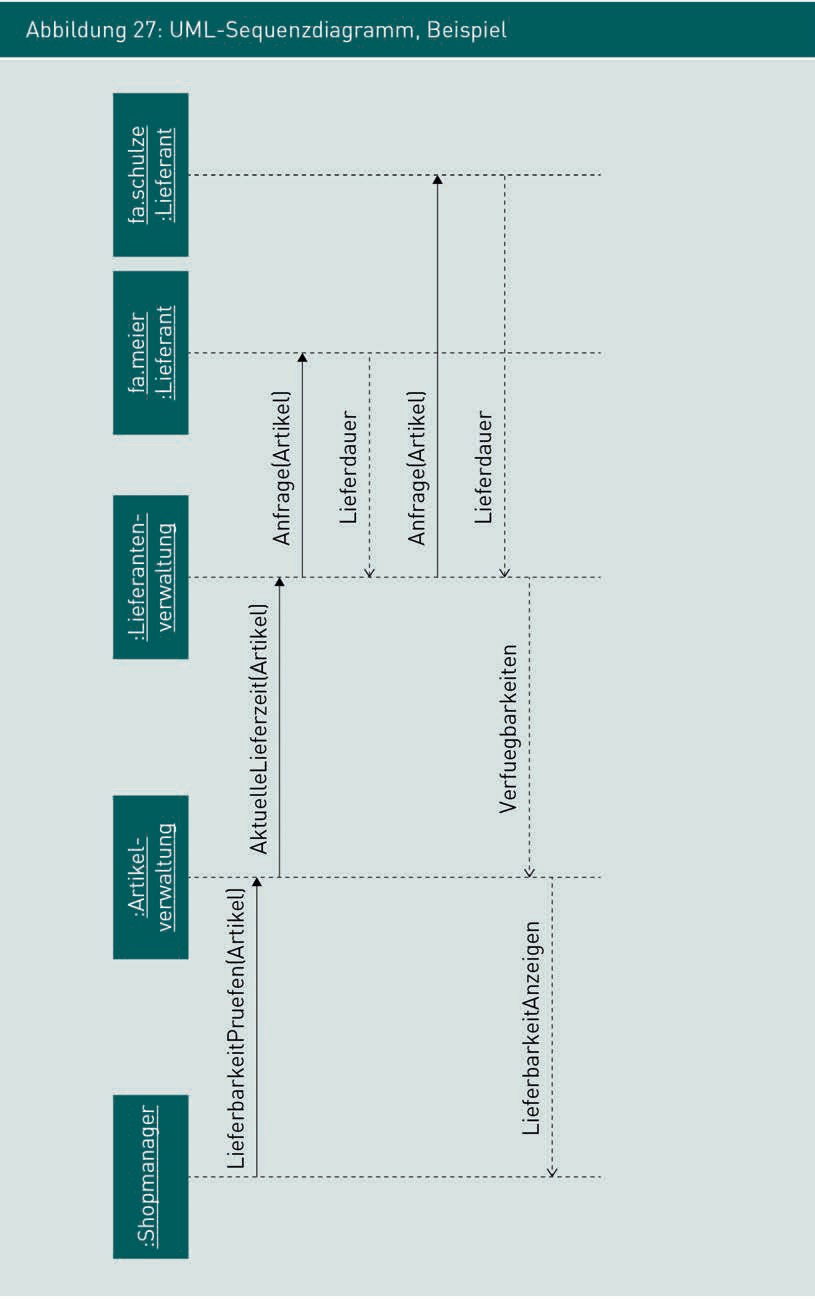
A UML sequence diagram (SED) is suitable for the detailed modeling of technical protocols between systems or components. While activity diagrams focus on the conceptual perspective, sequence diagrams can be used to model every detail of technical communication. When refining system interfaces, it is often necessary to incorporate technical details such as authentication, authentification, establishment of a suitable communication channel as well as status, test, and error messages.

The sequence diagram supports this by illustrating the interaction between communication partners over time, with due regard for the exchanged data structures.



The UML sequence diagram is a UML behavioral diagram (see Figure 26) which models individual operations. In this way, system behavior can be specified and illustrated using examples. Generally speaking, a sequence diagram behaves toward an activity diagram just as an object diagram behaves toward a class diagram: The sequence diagram illustrates a particular operation, while the activity diagram represents the sum total of all possible operations. Figure 27 illustrates a sequence diagram for the detailed specification of availability inquiries for items in an online shop. Table 17 explains the basic notation elements used in a UML sequence diagram.

Specification of technical system interfaces



|  |  |  |
| --- | --- | --- |
| Table 17: Basic notation elements used in the UML sequence diagram | | |
| Name | Meaning | Rendering |
| Interaction partner | An interaction partner, rendered as a box, is an actor who participates in the interaction. Interaction partners may include systems, users, classes, or specific objects. Interaction partners are labeled in the object notation: The typename is underlined and preceded by  “:”. If possible, a specific title is assigned to the object (in our example, shop manager). If a specific title cannot be given, it is simply preceded with “:”. |  |
| Lifeline | This vertical, dotted line without an arrow emanating from an interaction partner marks out their area of responsibility. |  |
| Call | A solid line with a solid arrowhead connects the lifelines of two interaction partners. The call models the fact that an interaction partner activates a function of another interaction partner or sends a message. The call can be labeled with the activated function and/or the sent message object. |  |

Specification of technical system interfaces

|  |  |  |
| --- | --- | --- |
| Name | Meaning | Presentation |
| Reply | A dotted line with an open arrowhead connects the lifelines of two interaction partners. The reply transmits the call reply to the activating interaction partner. The type of message and details of the content can be modeled on the arrow. |  |

The notation elements outlined above are suitable for denoting most common and relevant types of information. The sequence of interaction calls and replies is determined by the position of the arrows, when reading from top to bottom. Two arrows cannot be positioned on the same (notional) horizontal line in a sequence diagram, because it would then be impossible to decide which arrow should continue the communication. When creating a diagram, therefore, it is important to ensure that arrows are arranged across all lifelines in a clear sequence. The notation elements we have considered until now do not support the denotation of parallel operations or possible alternatives. In any case, this is not usually necessary because a sequence diagram generally models a specific operation.

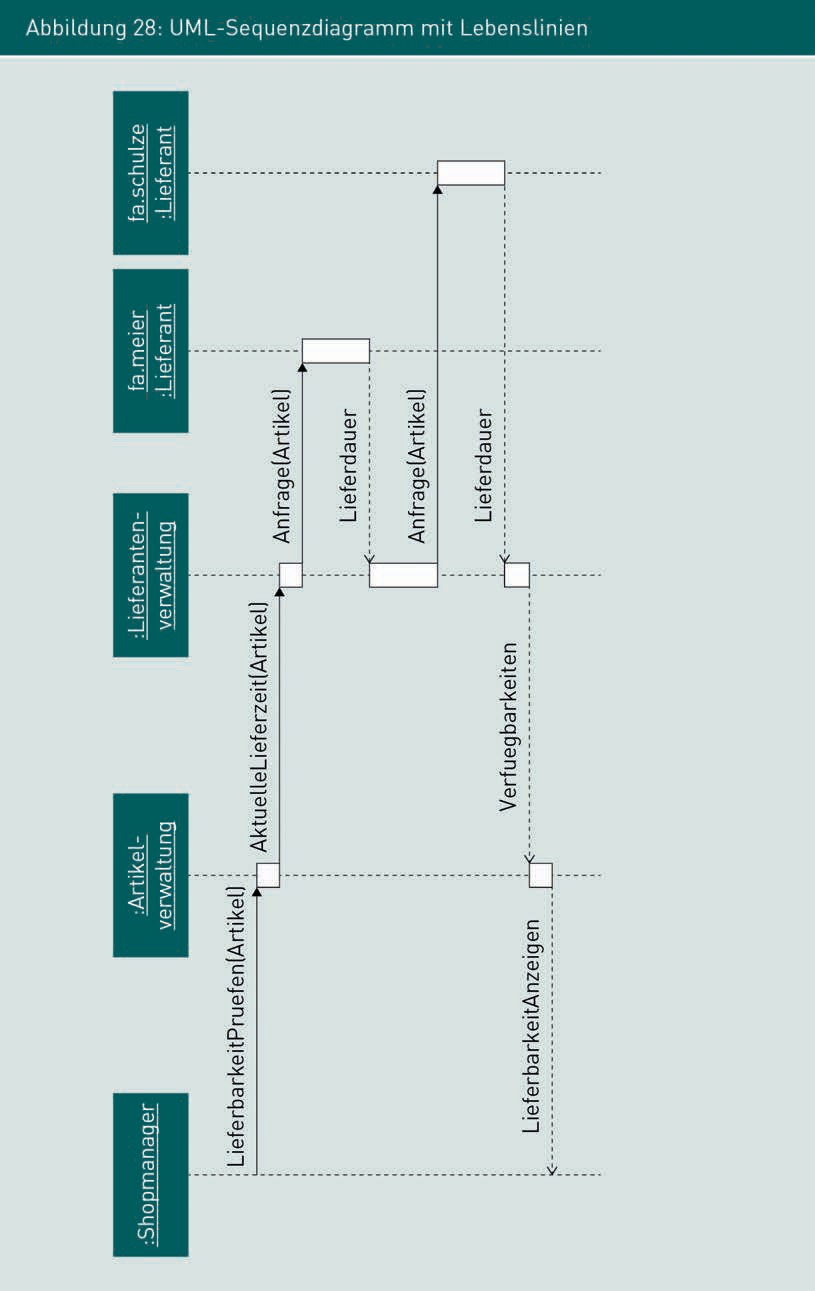
The statements derived from the sequence diagram concern the sequence of calls and replies only. A sequence diagram does not contain any information about particular processing times. The distances between the inquiry and the reply are not indicative of the time taken to process a query. Only logical information about the call sequence can be deduced from the diagram.

Other frequently used notation elements include activation bars (see Table 18) and combined fragments.

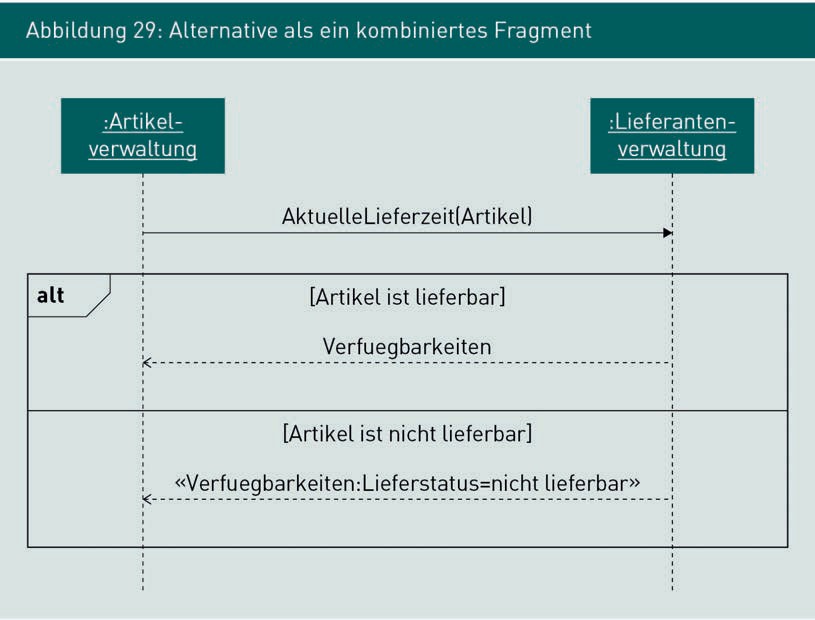
|  |  |  |
| --- | --- | --- |
| Table 18: Other important notation elements in a UML sequence diagram | | |
| Name | Meaning | Presentation |
| Activation bar | A white box on an interaction partner’s lifeline indicates when the interaction partner is active in relation to the currently displayed operation. |  |
| Combined fragment | Frame with type labeling in the top left, used to model options and alternatives in the sequence diagram. Possible types include parallelization, repetition, alternatives, and optional messages. |  |

Figure 28 shows our earlier example, this time with a modeled activation bar. Activation bars are optional and not compulsory. They are used to indicate who is currently working on a particular operation. The activity only refers to the operation shown in the diagram. For example, the “Inventory management” component cannot contribute to the current operation (between inquiry to and result from supplier management) and is therefore inactive for that operation. In the meantime, however, depending on the technical design of inventory management it may be involved in processing other inquiries not shown in this sequence diagram. The length of the activation bar is not indicative of the length of processing; larger activation bars in the diagram may well be faster than smaller ones.

Specification of technical system interfaces



In practice, combined fragments are also commonly used alongside activation bars. A typical example is shown in Figure 29, which models an alternative for the content of the reply message depending on the outcome of the supplier inquiry: If the item is in stock, a normal reply containing availability details will be sent, and if it is not in stock from any supplier, inventory management will set the delivery status to “not available”.



Using combined fragments significantly enhances the expressiveness of a UML sequence diagram. An unlimited number of interaction steps may be modeled in the individual sections of combined fragments, so theoretically, the content of any UML activity diagram could be denoted in a sequence diagram. However, the larger the combined fragments become and the more of them are used, the more confusing the model will be.

Note

The OMG of the UML sequence diagram contains numerous other notation elements used primarily for modeling reactive systems such as technical controlgear in air-conditioning systems.

Specification of technical system interfaces

However, at this point we will only consider elements commonly used in the specification of technical interfaces in information systems. UML sequence diagrams can additionally be used to model various call types and options / decisions relating to sequence control. These elements are described in detail in the specialist literature and in the standard OMG UML document.

In summary, the use of UML sequence diagrams is appropriate when

* Denoting complex interactions and interaction cascades between components
* The sequence of exchanged messages is crucial and
* Denoting procedural details for a special application, such as a particular path, in the form of an activity diagram.

Top tip:

Use the UML sequence diagram to illustrate specific operations and cascades within a system and across system boundaries. Use simple notation elements wherever possible and omit all information which is not strictly necessary.

###### Strength and weaknesses of UML diagrams versus behavioral specification

To conclude this section, Table 19 provides an overview of the strengths and weaknesses of individual UML behavior diagrams in the specification of technical interfaces.

|  |  |  |
| --- | --- | --- |
| Table 19: Strengths and weaknesses of UML diagrams when specifying technical interfaces between components | | |
|  | Strengths when specifying interfaces | Weaknesses when specifying interfaces |
| UML use case diagram | Denotes at the highest level that interfaces exist | No conceptual or technical details |

|  |  |  |
| --- | --- | --- |
|  | Strengths when specifying interfaces | Weaknesses when specifying interfaces |
| UML activity diagram | * Links conceptual operations with exchanged data structures * Overview of required messages * Denotes the purely conceptual level | * No technical details of the interface * No details of the message content |
| UML sequence diagram | * Depicts complex interactions and technical interaction cascades in their precise sequence * Links operation and data structure | * Focus on specific operations * Operation isolated from the conceptual use case * Can soon become highly complex and technical * Only readable by those with advanced knowledge of UML |

### Data structures at component interfaces

While module 4.1 focused on the specification of behavior, this module focuses on the specification of data structures at component interfaces. Generally speaking, UML class diagrams (as UML structure diagrams) are also suitable for the specification of data structures at component interfaces.

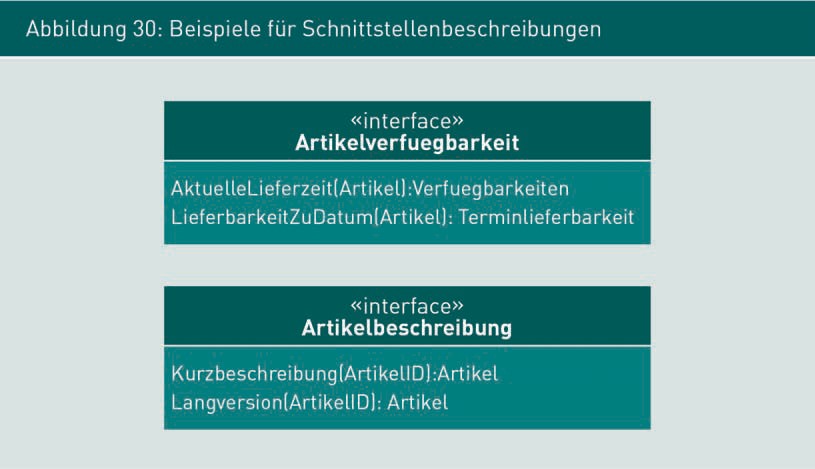
Before specifying the data structures, it is first necessary to identify the actual component interfaces. The modeled interfaces can be read directly from component diagrams (see Figure 24). In use case diagrams, the intersections between the communication relationship and the system boundary indicate an interface, while in activity diagrams, interfaces are indicated by the intersections between the control flow and the partition boundary. In every instance, technical interfaces must be given a unique, preferably meaningful name. For example, the interface names in Figure 24 allude to the subject matter. Interface names such as WK\_23\_C should be avoided.

Specification of technical system interfaces

###### Specification of interfaces

Once an interface has been identified, a detailed specification is needed, depending on the current project situation. Beginning with the component diagram which name every interface, the interface definition is then extended to include specific responsibilities.

In the simplest case, every interface is modeled in the component diagram by an appropriately named interface, together with the required messages and result types. Figure 30 shows UML interface specifications for the inventory management component from Figure 24. Pre-existing UML activity diagrams and sequence diagrams can help to identify the required functions, their parameters, and their return types. A suitable interface function must be allocated to each call in the sequence diagram. The messages rendered as object nodes in the activity diagram are modeled as parameters or return values of functions in the interface description. You cannot specify fewer parameters and results in the interface description than are found in the sum total of existing UML diagrams.



Unlike the specification of object-oriented classes, component interfaces serve to elucidate responsibilities. The software architect is responsible for determining whether there is a specific implementation for each method in any given class. Components are expected to offer functions, but their internal implementation is not relevant from a specification perspective.

###### Specification of data structures

Once the specific functions and associated messages have been identified, the next step is to specify the message structure. As the project progresses, the software architects and developers draw on the specified data structures as the basis for reconstructing the business logic of communication between components and/or between systems. The specification of data structures includes a detailed definition of the information available to the component and how it is structured. Figure 31 shows an excerpt derived from data objects from the item availability interface: The data type item and the data type availability can be derived directly from the interface. The attributes of these data types are derived, firstly, from the requirements already documented, and secondly, from the interface descriptions of existing components. Industry standards, guidelines, and corporate policies are often used as sources for the specific attributes of interface data structures.

Specification of technical system interfaces



In Figure 31, data structures are specified in the form of UML class diagrams. Additionally, information about interfaces can usually be given in the form of a table, text, or entity relationship diagram. However, as the UML class diagram notation may be integrated into other types of diagrams, we recommend the conceptual/technical specification of data structures in the form of class diagrams as they are capable of modeling the most common data structures, and specific interface formats can often be derived directly during downstream development phases

Entity Relationship

Diagram A type of diagram used for the visual modeling of data structures, often in a database environment.

The specification of data interfaces will produce at least one interface description for each technical interface in the component diagram with all the required functions and data structures, together with class diagrams specifying the particular data structure transmitted with each message.

Summary

Specification must include a detailed description of the messages exchanged between components and systems at technical interfaces. The UML activity diagram plus object node and the UML sequence diagram are suitably structured for documenting which messages are exchanged between systems and when. While the activity diagram focuses on the conceptual perspective, the sequence diagram supports the modeling of technical communication details. In this way, technical details such as authentication, authentification and the establishment of suitable communications channel as well as status, test and error messages may also be specified when refining system interfaces. The sequence diagram shows the interaction between communication partners over time with due regard for the exchanged data structures. The UML class diagram is suitable for the specification of data structures at component interfaces. Once the component interfaces and their functions have been identified, the particular data structures are described in detail, including a clear definition of which component information is available and how the information is structured in detail.



# Unit 5

## Specification of detailed conceptual data models

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... where to use conceptual/technical data models in SRS

... which characteristics can be used to extend and refine conceptual data models

... how to test data models.

DL-D-ISPE01-L05

1. Specification of detailed conceptual data models

### Introduction

Detailed data models are usually required both to describe business objects processed within systems and to specify interfaces. The business objects and their properties have already been identified and documented during the course of conceptual requirements engineering. This unit focuses on the conceptual/technical detailing and refining of data models at a highly technical level as an essential aspect of the specification of technical interfaces.

### Conceptual data models and their application areas

Conceptual data

models Models used in the specification of conceptual entities (business objects) and their

correlations.

Conceptual data models are used to uniformly define conceptual classes (business objects) and their correlations for all participants. Software models (such as UML class diagrams) serve to specify the structure, composition, and relationship between different functional classes. With complex systems that process identical data in different components, conceptual data models are a way of ensuring that exchangeability between individual components and systems is retained. For example, the data entered via a GUI must observe the data model in the same way as data stored in the database.

A conceptual data model therefore acts as a parenthesis around the different aspects of a system and its specification.

###### Data models for component and system behavior

When specifying component and system behavior, data models (also known as domain models) are used to document business objects, their properties, and their correlations. The modeled conceptual classes clearly and concisely denote which information the system must process and/or support. While process models prescribe the HOW of a system, data models specify the WHAT.

From the development team’s perspective, the conceptual data model provides a rough framework for defining the system’s data architecture. It can also help to identify requirements by acting as a kind of checklist: Each relevant element in the data model must be supported by the software system. The development and detailing of the conceptual data model in collaboration with the relevant stakeholders is vital for transferring knowledge from the requirement sources to the development team.

Specification of detailed conceptual data models

###### Data models for GUI specification

An information system communicates with its users via the user interface, so the GUI is used for manual inputs and for modeling the business objects stored in the system. All relevant elements from the domain model must therefore be taken into account when designing the GUI. The structure of business objects also has a decisive influence on the design and implementation of the system interface, because many of the GUI requirements are derived directly from the conceptual data model. Examples include the number of input and output elements, requirements governing validation via the data type, and the input of self-selected values via the text box or from a list.

###### Data model for technical interfaces

A user can extract the key information from a letter or email and enter it in the system via the GUI, thereby transforming an unstructured message into structured information. Software systems communicate directly at technical interfaces via the exchange of messages. Unlike letters and emails, the data transferred at a technical infrastructure must conform to a strict predefined structure, otherwise the message will not be (fully) understood. A conceptual data model provides the basis for a detailed specification of messages at a technical interface. Smooth interoperability between software systems unless the development team reaches a detailed, thorough agreement on the structure of messages exchanged between systems.

###### Documentation forms for data models

Conceptual data models can usually be documented with the same concepts as technical models. Visual software models such as UML class diagrams, UML object diagrams, and entity relationship diagrams are widely used. Data models at interfaces are often described using XML as structured text. Table 20 compares some typical documentation forms and their application areas.

|  |  |  |
| --- | --- | --- |
| Table 20: Typical documentation formats for data models | | |
| Documentation form | Description | Typical applications |
| UML class diagram | UML structural diagram; uses range from analysis to object-oriented design of classes | For documenting conceptual and technical elements at type level, from analysis through to implementation |
| UML object diagram | UML structural diagram; represents specific instances of class diagrams | For representing specific data records or business objects; for illustrating instances in a class diagram |
| Entity relationship diagram | Structured, visual representation of entities, their attributes, and relationships; may be modeled directly in database tables | For specifying data and database models; often used in a database context |
| XML | Structured, text-based description of data models that can be read by both humans and software systems | For specifying data models at system interfaces; for specifying data models of documents and strict tree structures |

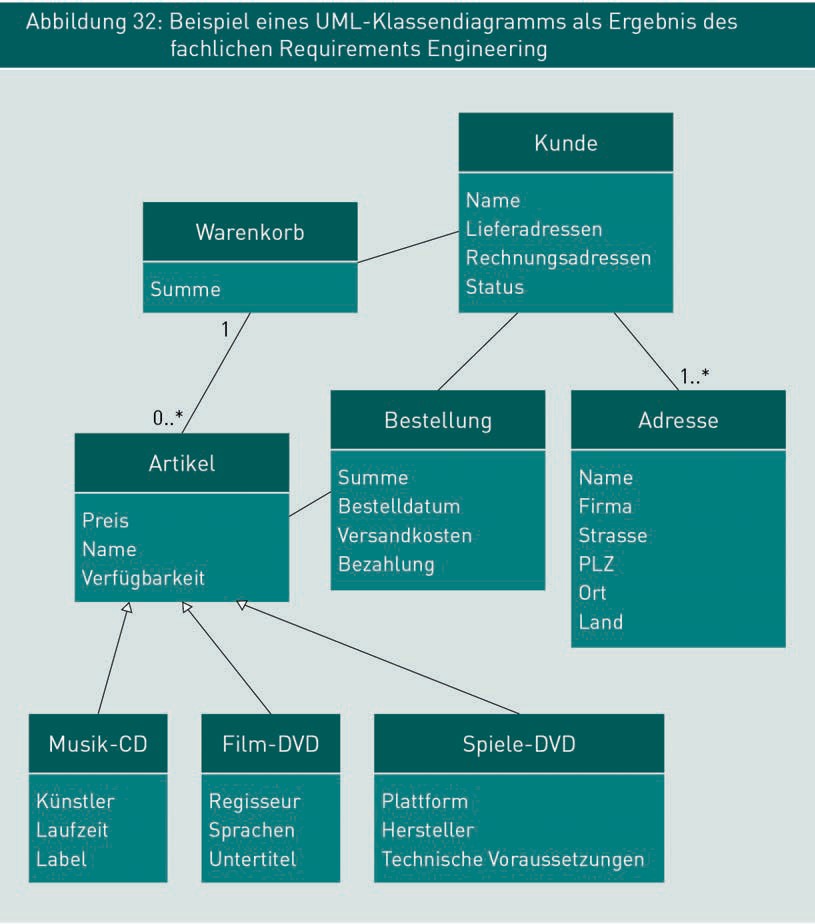
### Detailing the UML class diagram

Conceptual entities Industry-specific, real and virtual “things” which are managed by information systems.

In a requirements engineering environment, class diagrams are used to document static concepts in an application area. This includes business objects, conceptual entities, persons, objects, and systems and their relevant properties, relationships, and interdependencies. Their purpose is to document, comprehend, and communicate the conceptual problem. The classes in the class diagram are therefore interpreted as concepts of their environment rather than elements of a system.

Specification of detailed conceptual data models

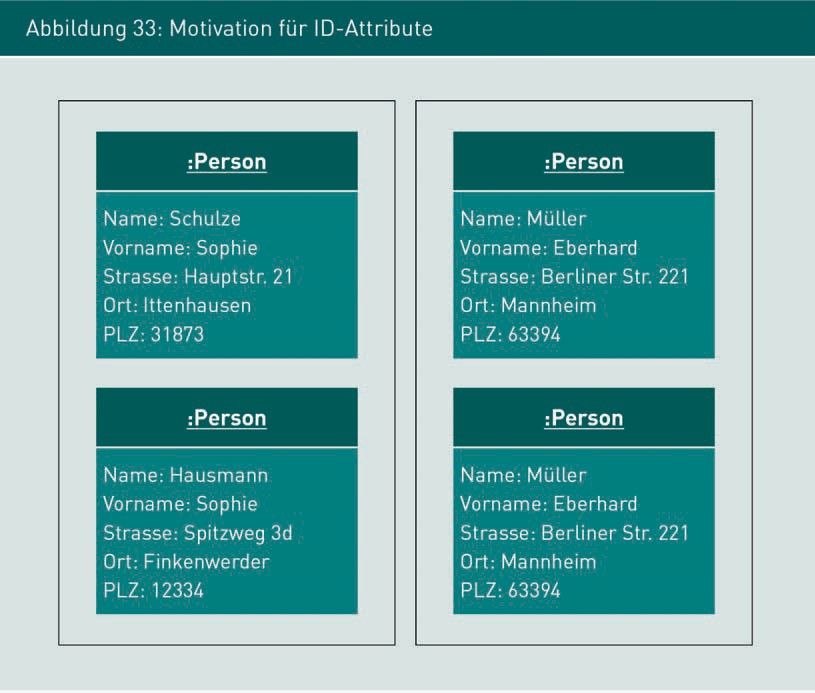
A UML class equates to a conceptual concept, i.e. a group of objects with the same characteristics, such as customer, item, or order in Figure 32. The class diagram is not suitable for modeling behavior or operations.



Below, we consider some common aspects of the conceptual/technical specification of data models. In each case, we examine which information is needed to support the functions required by the system. This list of aspects is not intended to be exhaustive, and merely represents some frequently asked questions.

###### ID attribute for unique identification

Business objects must usually be clearly distinguishable from one another and uniquely identifiable. Figure 33 illustrates two scenarios each with two specific objects where personal data is stored. Do the data records on the left refer to two different people or the same person whose name has changed due to marriage and has moved house? Do the data records on the right refer to the same person or are they two different people who just happen to live in different flats in the same building?

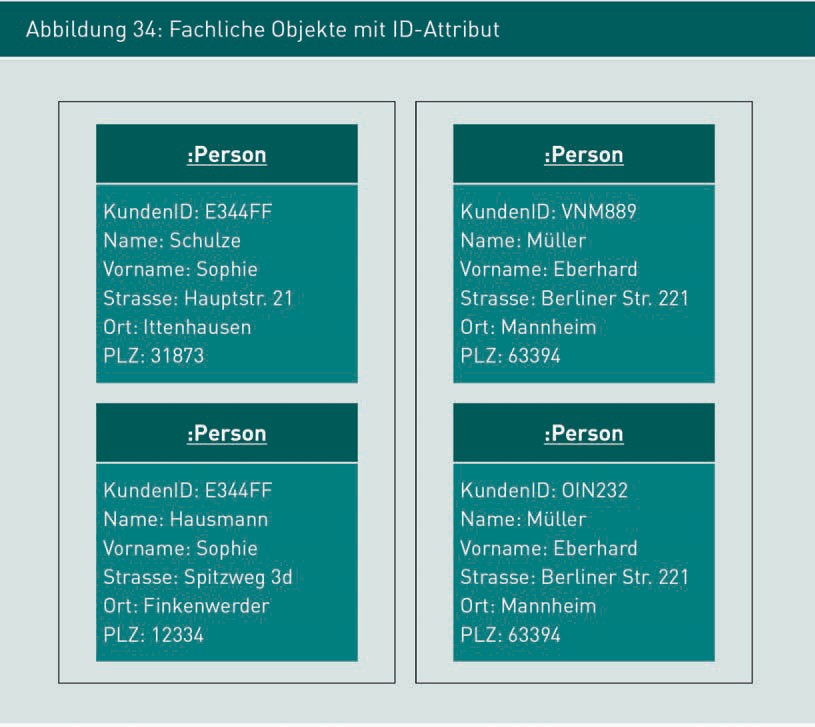


The conceptual data records do not provide a definitive answer for either of these questions. When referring to an instance of a class in an object-oriented system, two objects are considered identical if all their attributes are identical. Viewed from this perspective, the two objects on the right would be identical while the two objects on the left would not.

The addition of an ID attribute to the data model allows data records to be distinguished and identified even if the business object’s conceptual attributes are insufficient for this purpose.

Specification of detailed conceptual data models

Figure 34 illustrates instances of a data model which has been enhanced with an ID attribute to provide the answer to the above identity questions.



These conceptual ID attributes will not usually change over a business object’s lifecycle. A change to the ID implies a conceptually different object.

There is a crucial distinction between conceptual IT attributes and technical ID attributes. Conceptual ID attributes must never be used for the technical distinction of data records or business objects, for example as an entry in a database. It is often the case, when upgrading an outmoded system, that the conceptual data is transferred into the new system. This may necessitate an amendment to the value ranges of the admissible technical IDs. As such, it must be possible to amend the technical ID independently of the conceptual ID attribute. Conversely, it must also be possible to amend a conceptual ID without simultaneously amending the technical ID. There is a risk that extensive amendments and modifications may be needed if this distinction is not consistently maintained.

It is also possible for data records to be given a separate, internal technical ID which is not clear outside of the system. Comparisons at system interfaces should therefore only be made on the basis of one or more conceptual ID attributes.

###### Completeness of all attributes

When preparing a conceptual/technical specification, it is important to ensure the completeness of all relevant business objects and their attributes. For data models at technical interfaces in particular, it is crucial to ensure that both the message sender and its recipient are using the same information structure. Previously modeled operations (see module 3.3) and scenarios (see module 4.1) may be useful when verifying completeness; the same also applies to object diagrams (see module 5.3).

###### Detailed properties of attributes

Whereas the documentation of attributes and their name is often sufficient for the purposes of RE, conceptual/technical specification dictates that a data type must be specified for each attribute. Among other things, definition of a data type will influence:

* The choice of input and output elements on the GUI
* The validation and conversion rules for the GUI
* The chosen technical data type in the database and all other technical components required for interim storage and editing; and
* The chosen technical data type for messages to system interfaces.

The following template can be used for the detailed and complete specification of attributes, whereby the elements in “[]” are optional:

*Name of attribute: Data type [Multiplicity] [= Default value] [{Property values|Constraints}]*

String Technical data type for storing strings of characters of unlimited length and type.

Data type

A string is a very general data type for attributes and should be chosen if there are no conceptual or known technical restrictions on the value range.

Consequently, the value ranges for attributes should only be restricted for conceptual or known technical reasons. Length restrictions and limitations on accepted values are often specified to save storage space and facilitate GUI inputs. These unnecessary restrictions may prove impractical over the application’s lifecycle, leading to avoidable modifications and maintenance work. A typical example is the specification of attributes when building numbers are saved in the form of integers (whole numbers), which would preclude the addition of building number 15b.

Specification of detailed conceptual data models

Another example would be to specify the data type for post codes as integers, so that when the postcode 01069 is entered, the leading zero is omitted and stored in the database as 1069.

Multiplicity

Optional multiplicities allow you to specify how many values may be linked to an attribute. Multiplicities are often modeled together with associations. In the UML class diagram, these quantities may be documented immediately after the data type.

Default setting

The optional default value setting (also known as an initial value) allows the definition of a value which is automatically pre-assigned to the attribute. For example, today’s date may be pre-entered in the date box by default, or a frequently selected option assigned to selection boxes. Default settings help to make a system more usable.

Property values

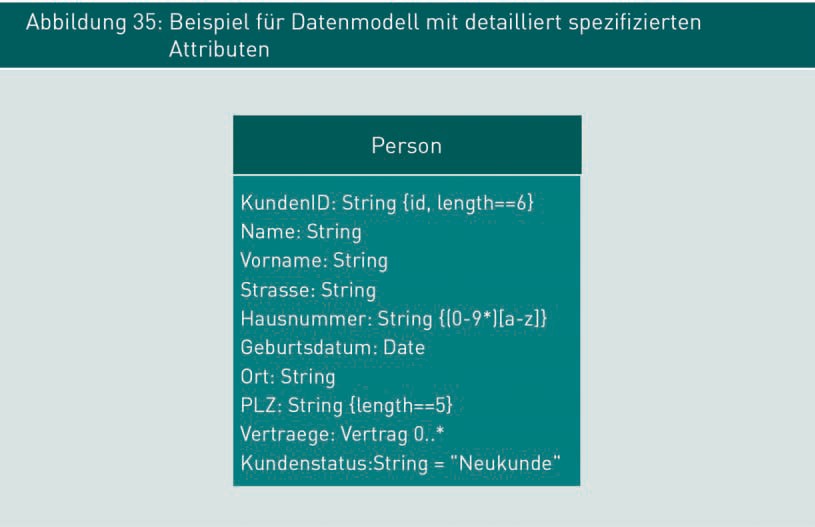
Optional property values are used to document additional information about the attribute that may be relevant when creating the system. Examples of property values include:

* {readOnly}: Value cannot be overwritten;
* {unique} or {id}: Value must be unique; no other object may have the same value for this attribute; or
* {ordered}: Values are listed in order and do not contain any duplicates.

Property values are specified as required on an individual project basis.

Constraints

Optional constraints (conditions, restrictions) are used to provide further details of admissible attribute values in addition to data type. Constraints can define the length and type of admissible characters in the attribute value, stipulate compliance with certain format guidelines, and define interdependencies with other attributes. For example, in the terms of a contract, one requirement is that the start date must be before the end date. Where these constraints are already known at the time of specification, they may be incorporated directly into the data model for that attribute.



Unlike object-oriented analyses, specifications do not document an attribute’s visibility.

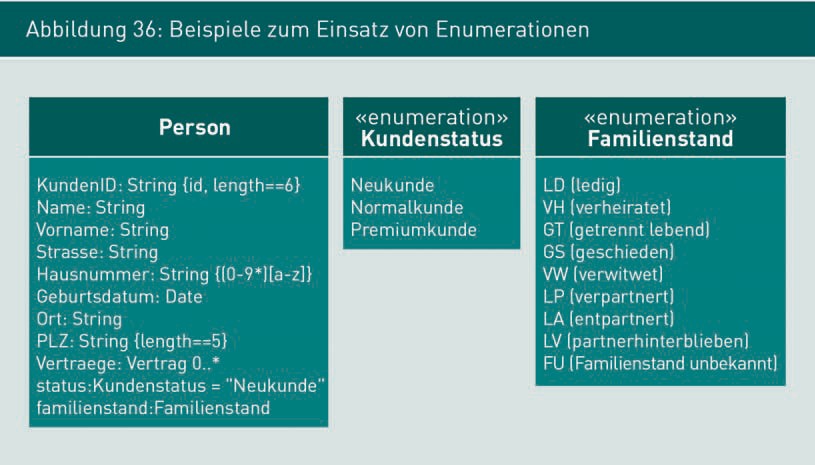
###### Enumerations: Predefined value sets

Some attributes in the data model will generally have a pre-defined set of admissible values. This means that an attribute’s specific value cannot be freely assigned within the value range for its data type but must belong to the predefined set.

When specifying a conceptual/technical data model, these types of lists may be defined as a separate data type, known as enumerations. An example of the use of enumerations can be found in Figure 36. The list of potential customer statuses and the list of possible values for the attribute “marital status” are both outsourced to separate data types. Enumerations are specified in the same way as classes with attributes: The stereotype

<<enumeration>> denotes it as an enumeration data type rather than a business object. Possible values are specified as attributes of the enumeration class. Enumerations can be used in all other classes of that model. If you need to modify the value set of an enumeration type, you only need to do so once.

Specification of detailed conceptual data models



###### Constraints: Business rules on the structure of data models

In module 3.3 we examined the business rules for specifying component behavior. Structural business rules may be used for the detailed specification of data models. These rules specify key static aspects of a system which must apply to the data model at all times.

Here are a few examples of structural business rules:

* “A customer may apply for a partner card for exactly one life partner and a minimum of one and a maximum of three children.”
* “A customer with an average order value of >=€2,500 is awarded the status premium customer.”
* “The monthly contribution is always one-twelfth of the insurance premium.”

Structural business rules are comprised of terms and facts. Terms are words or descriptions with particular conceptual relevance in the data model, such as individual, partner card, or premium customer. These terms should be explained in a glossary. The facts describe the relationships between terms. Unlike business rules which specify behavior, structural rules relate only to the data model. Some of these rules are readily denoted in UML. The first example shown above can be modeled in a class diagram with relationships and multiplicities. Rules that cannot be modeled directly in the class diagram, such as the second and third examples above, may be documented in natural language to complement the data model.

Always write structural business rules in a declarative (descriptive) way, stating simply that they apply but without any indication of when the specified rules are reviewed. The development team decides when and where in the system this occurs.

###### Relationships between classes

Functional dependencies and relationships between classes are described as so-called relationships (or associations) between classes. The notation elements in the UML class diagram already used in requirements engineering are generally sufficient for a detailed specification.

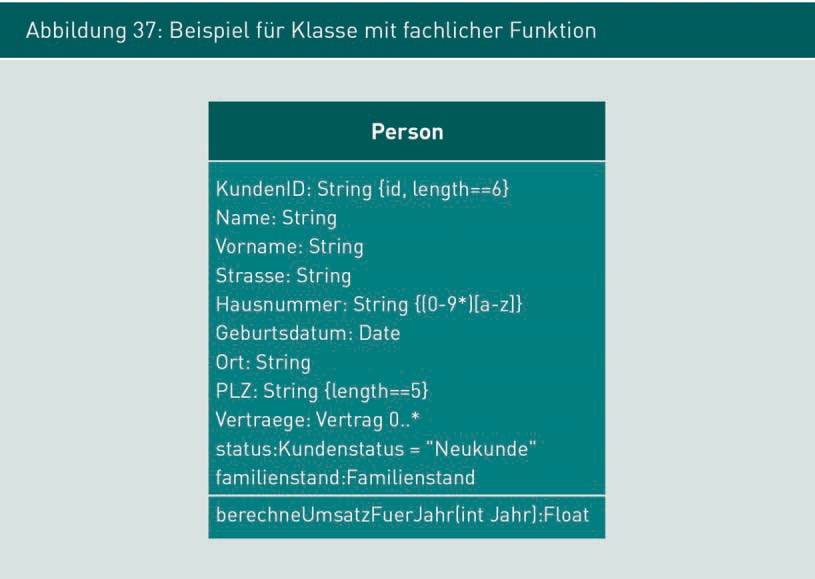
Note

If you would like to revise notation elements for modeling relationships between classes, please refer to the relevant modules in the Requirements Engineering course.

###### Specification of functions

Occasionally in a conceptual data model, classes may be identified which are responsible for key functions, such as the shopping cart for calculating the total invoice amount, or the premium calculator for working out complex insurance premiums, or a credit calculator to work out the costs and interest of a consumer loan in direct banking. Responsibilities for conceptual functions can be included in the data model as methods relating to modeled classes. However, only essential conceptual functions should be specified at this point. Trivial and obvious methods, such as reading and amending values, should not be modeled here. For function, alongside the name it is also important to describe the quantity and data type of the required parameters together with the quantity and data type of the return type(s).

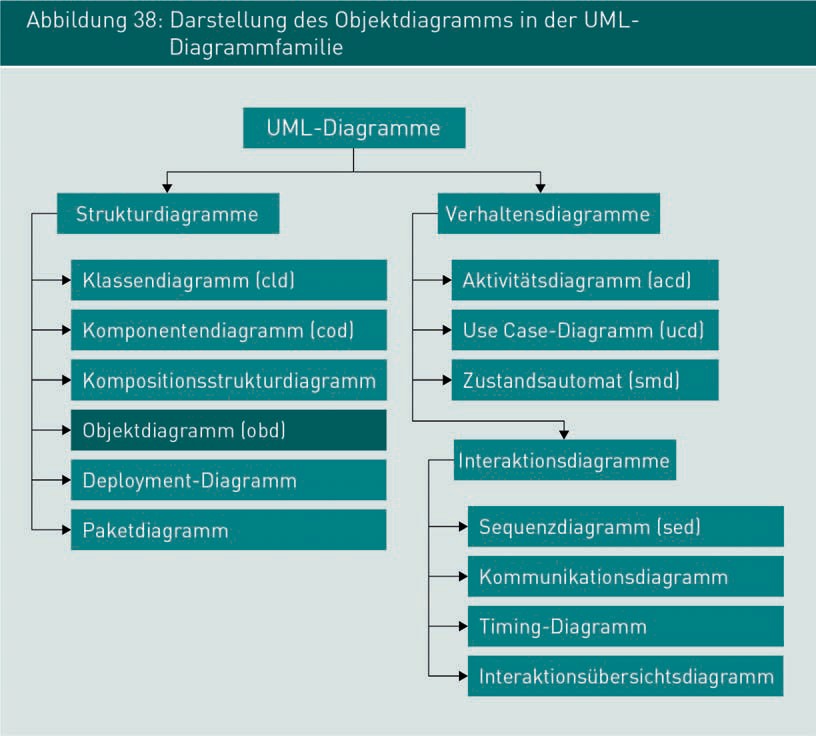
Specification of detailed conceptual data models



### Using UML object diagrams to check class diagrams

As we have learned in module 1.3, UML object diagrams can be used in the detailed specification of data models to check the appropriateness and completeness of class diagrams. Object diagrams are a type of class diagram, used to represent an instance of a class by modeling objects whose attributes contain specific values. The persons denoted in Figure 34 are examples of specific objects in a “person” class.

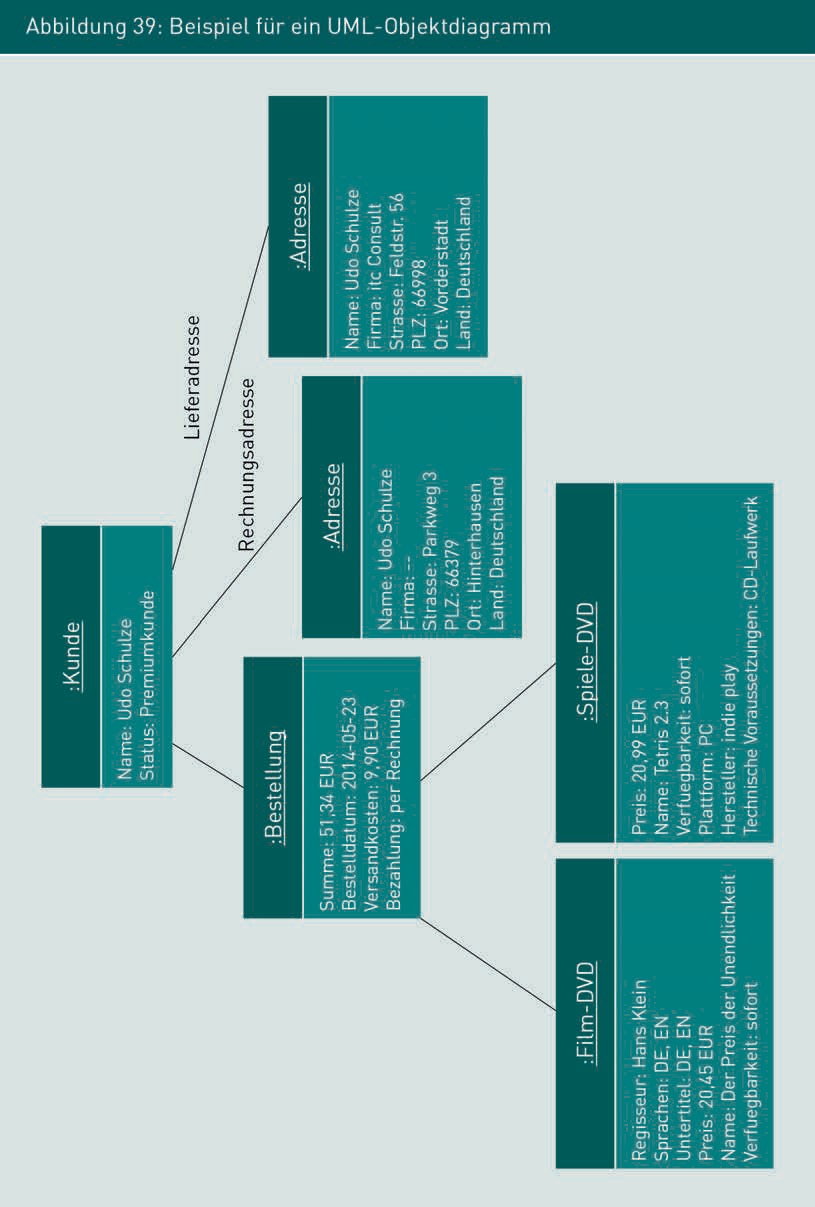
Objects differ from classes in that they show a particular value for each attribute of an object. The class name is underlined and preceded by a colon “:”. Where necessary, the object can be identified with a unique ID to the left of the colon. The ID in the object header is a technical ID, rather than a conceptual ID, which is only valid in the object diagram.



Certain complex system states may be denoted using object diagrams in order to check and evaluate the conceptual correctness of class diagrams, as well as the current status of data records. Object diagrams are particularly ideal for denoting quantities. By modeling quantities as multiplicities of relationships in the class diagram, each individual object is explicitly represented in the object diagram.

Modeling data records at instance level, rather than at type level (as in a class diagram) requires far less abstract reasoning than would be needed to comprehend a class diagram. Figure 39 illustrates a UML object diagram as a specific manifestation of the UML class diagram shown in Figure 32.

Specification of detailed conceptual data models



###### Key notation elements in a UML object diagram

|  |  |  |
| --- | --- | --- |
| Name | Meaning | Denotation |
| Object | Instance of a class; specific attribute values appear after the attribute name; anonymous objects (those without an object ID) are used if the object name is irrelevant in the current situation. |  |
| Relation-ship between objects | Relationships between objects are shown as lines; the attribute name is modeled on the line. |  |

To aid legibility and comprehension of the object diagram, the corresponding class diagram must also be available so that the current status of objects can be seen in context. There is no need to specify all values of all object attributes in full. Object diagrams are often used to illustrate or discuss very specific value constellations or statuses.

Specification of detailed conceptual data models

When creating an object diagram, you should only include those values which are currently relevant for comprehension and omit all other values.

###### Using object diagrams to check the correctness of class diagrams

Class diagrams contain numerous notation elements which can be helpful when specifying data models, but their interpretation requires knowledge and practical experience. Because attributes and relationships are inherited and quantities are documented as multiplicities, the conceptual accuracy of detailed class diagrams cannot always be verified with all stakeholders.

Class diagrams project a slice of reality (the real world) at type level by describing (summarizing) multiple similar real-life objects as types (classes). Object diagrams may help a layperson to verify the validity of this projection when modeling. The interaction between the UML object diagram in Figure 39 and the corresponding UML class diagram in Figure 32 illustrates this issue. For example, whereas in Figure 32 the legacy relationships are conceptualized, Figure 39 shows each actual object and each actual link between those objects.

Wherever possible, use real-life data records to check class diagrams. For example, work with a subject expert to select typical business objects from their database then generate them in the form of an object diagram for that particular class diagram. If a corresponding object diagram can be generated for each relevant data record, this indicates that the specified data model is probably suitable. Often, object diagrams will identify gaps and weaknesses in the data model which then need to be rectified.

Top tip:

Always use realistic data to generate representative examples! Get subject experts to supply you with real-life data when checking data models. If this is not possible for data privacy reasons, try to make your test data as realistic as possible.

Once a class diagram has been modified, it can and should be checked, for example when maintaining and upgrading an application. Modified class diagrams are checked in exactly the same way as newly created diagrams.

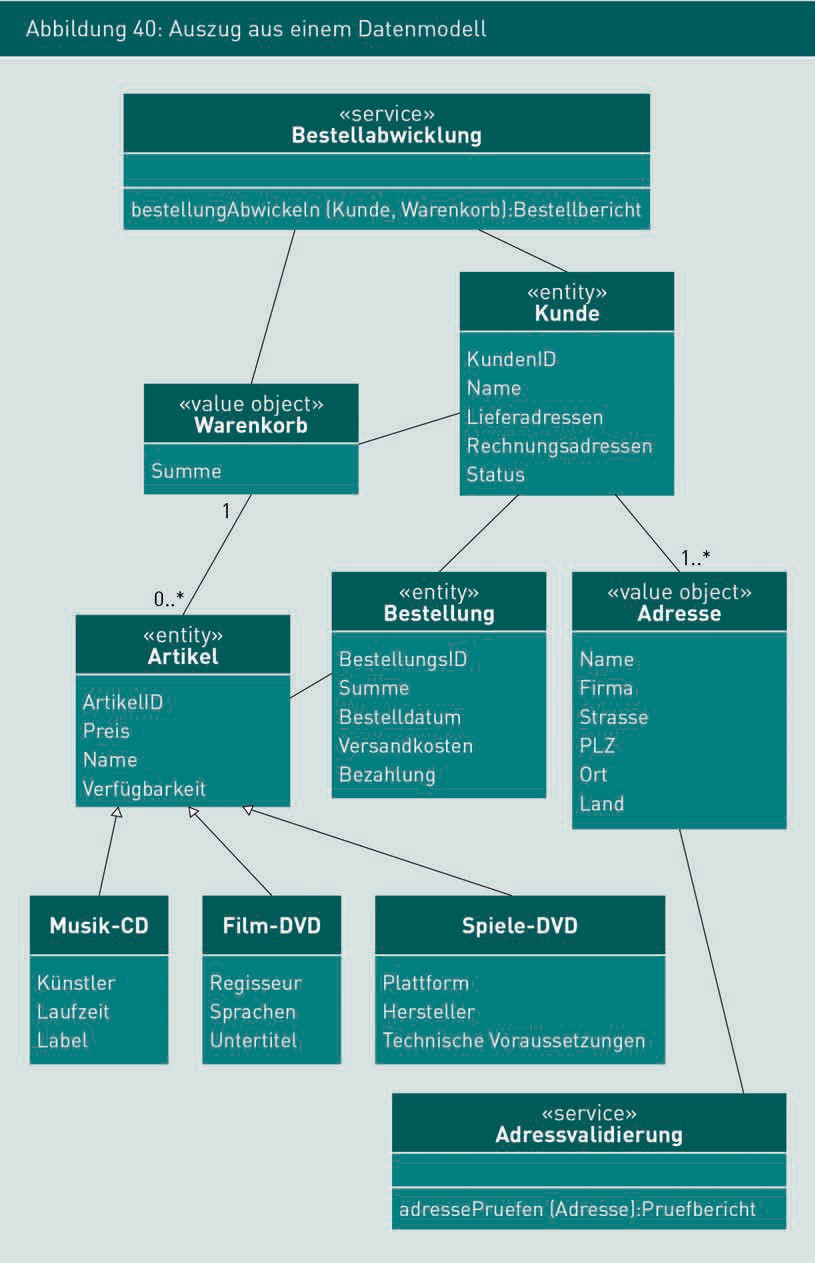
### Typical elements in conceptual data models

Detailed conceptual data models contain numerous model elements, and as a result, they are often quite large and complex. While potentially confusing, they also allow you to focus selectively on key relevant areas.

The categorization of model elements can make data models easier to handle in a more selective way. The elements of a conceptual data model can generally be classified into one of the following three categories:

* + Entities
  + Values (also known as value objects) and
  + Services.

Specification of detailed conceptual data models



###### Entities

Historization A record of all changes imple-mented, often showing the date of the change and the person who made it, so that all changes can be precisely tracked.

Entities are elements of a data model which have a conceptual identity. Entities are often subject to a lifecycle that can be described with a state diagram. An entity’s characteristics change over the course of its lifecycle. An application for insurance is one example of an entity. It is subject to its own lifecycle, during the course of which it evolves from an application to a policy. At the same time, the quantity and nature of the business object’s relevant attributes also change. Giving entities a conceptual identity can be helpful for tracking them throughout their entire lifecycle (see also module 5.2). A comparatively long lifespan is also a typical property of entities. Historization and traceability are among the requirements typically placed on entities, enabling every previous state of an object to be reconstructed.

When preparing a data model, be sure to assign a conceptual ID attribute to each entity. Keep the number of attributes modeled on an entity to a minimum. Any information not required for key functions can be outsourced into value objects. In the class diagram, add the stereotype <<entity>> to denote a class as an entity. Figure 40 denotes the classes “customer” and “order” together with all items as examples of entities.

###### Value objects

Value objects are data objects without their own identity. Only the data stored in these objects is conceptually relevant. They do not have their own lifecycle, nor do they have a conceptual ID attribute. Value objects can be used to store additional information about entities, among other things. Conceptually, value objects are “less interesting” than entities. Unlike entities, value objects can be copied directly without risking conflicts with ID attributes.

A value object is a conceptual grouping element. In other words, modeling should ensure close conceptual links between the attributes in a value object. Multiple value objects may be modeled for a given entity. Value objects are denoted by the stereotype <<value object>>. In Figure 40, the “Address” class is an example of a value object. Other typical examples might include bank accounts, order items, or communication data.

###### Services

Services are stateless functions not directly attributable to entities or value objects, where “stateless” means that the service itself does not have any attributes and hence no inner status. Stateless functions with identical parameters will always deliver the same result. If functions in a conceptual data model are identified which cannot be directly assigned to classes with attributes, these may be modeled as services.

Specification of detailed conceptual data models

A service is only described by its behavior, as there is no data stored in it. When modeling services, remember to ensure that the parameters and the return value all originate from the set of entities and value objects. In data models, services are denoted with the stereotype <<service>>. The classes Order Processing and Address Validation are examples of services in Figure 40. Both of them are classes without attributes, each with a defined specialist function.

By classifying the elements in a conceptual data model into entities, value objects, and services, you can help to keep the data model manageable, while at the same time providing valuable information for the software architect and development team. Depending on the category allocated to a data model class, various draft decisions will be made over the subsequent course of the software project.

Note

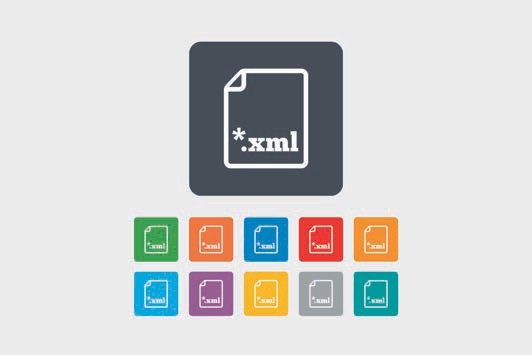
Read more about this topic in the book Domain-Driven Design by Evans (2003).

The definition and use of stereotypes in UML models is very clearly explained in Rup/Queins (2012).

Summary

Conceptual data models are used to specify functional classes (business objects) and the correlations between them in a uniform format for all participants. They represent a further refinement of the models documented in conceptual requirements engineering. Relevant aspects include ID attributes, attribute characteristics such as data type, default value and constraints, as well as enumerations. UML object diagrams can be used to check data models for conceptual accuracy. Elements in the conceptual data model can generally be assigned to one of three categories: entity, value object or service.

In the class diagram, this is denoted by stereotypes.



# Unit 6

## Using structured text in the specification of data interfaces

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... how to structure XML documents.

... how to specify the structure of XML documents with XML schemas.

... how to generate class diagrams from XML documents.

... how to structure the XML specification of Web services.

DL-D-ISPE01-L06

1. Using structured text in the specification of data interfaces

### Introduction

As well as using software models and texts for specification, messages at technical interfaces are specified as structured texts which are readable both by humans and machines. In this unit, we examine eXtensible Markup Language (XML), a standard used worldwide.

### Structured text as an interchange format

UML class diagrams (see module 4.2) can be used for the detailed specification of data structures at technical interfaces. However, at technical interfaces, it is particularly vital to ensure that the actual messages exchanged are clearly defined and conceptually accurate in terms of their nature and structure. This is achieved by converting the specified data structure into a particular exchange format. The following requirements must be met when selecting and defining message formats for the storage and transmission of data: The data structure must be maintained, it must be easily readable by humans, and information must be stored efficiently.

Possible exchange format categories include:

* Binary messages
* Unstructured text messages and
* Structured text messages.

###### Binary messages

In a binary message, data is exchanged between systems in the form of a sequence of 0s and 1s. The technical data structure is retained and can be reconstructed directly. The structure and content of binary messages cannot be directly read or written by humans, but data storage is usually very efficient.

Example

Serialization of Java objects using the class java.io.ObjectOutputStream. It converts objects and their state into a byte stream. The class java.io.ObjectInputStream allows Java objects to be re-created.

Specification of data interfaces with structured text

###### Unstructured text messages

Unstructured text messages use typical text characters with no generally prescribed format. The basic structure of the message cannot be defined. When these messages are processed by a system, rules must be agreed upon, but it can be time-consuming to verify compliance. Unstructured text messages are easily read and written by humans, but the technical data structure is lost. There is a risk of syntax errors, i.e. violations of the agreed message structure.

Examples

Files with the extension TXT, short messages like tweets or text messages, and text in the error messages from a Java exception.

###### Structured text messages

Structured text messages must be given a clearly defined and readily verifiable structure. The system checks and processes these messages directly based on the specified structure, so that quality is readily guaranteed. The technical data structure of the message is converted into a text structure, which in turn can be converted into a technical data structure using defined rules. Structured text messages can be read and written by humans with due regard for the defined structure. Possible syntax errors are easily identified because the correct format of each message is automatically checked prior to processing.

Example

XHTML for websites, XML for all types of documents and messages, JSON for the transmission of objects and their attributes.

XML languages are very widely used as an exchange format in industrial environments, so the following module will focus on the definition and use of XML languages. The JSON format is also widely used, often for the exchange of objects between browser and web applications.

### Structure and layout of XML documents

XML (eXtensible Markup Language) is a language standardized by W3C for storing structured data in text files. It facilitates the storage of information using hierarchically structured elements and their attributes. However, the names of the elements and attributes are not taken from a specific vocabulary but can be freely defined using certain rules.

XML

A widely used, standardized language for storing structured data in text files and exchanging it between systems.

For example, a simple XML file with item data could look like this:

<?xml version="1.0" encoding="UTF-8"?>

<order>

<headerdata>

<orderno>48729</orderno>

<customer>

<title>Mr</title>

<firstname>Samuel</firstname>

<surname>Berg</surname>

<birthname/>

</customer>

<order\_dated>2011-04-12</order\_dated>

<customerno>AD-4333 532</customerno>

</headerdata>

<positiondata>

<position>

<id>1</id>

<itemno>57823566</itemno>

<quantity>4</quantity>

<priceEUR>15.90</priceEUR>

</position>

<position>

<id>2</id>

<itemno>54733462</itemno>

<quantity>2</quantity>

<priceEUR>8.90</priceEUR>

</position>

</positiondata>

</order>

Each XML file has precisely one single root element (in this example: order) which encloses all the other elements (in this example: headerdata and positiondata). In our example, the header data contains general information about the order while the position data contains a list of order items with the position ID, an item number, the quantity ordered, and the price.

The names of the XML elements are shown between pointed brackets (“<” and “>”) and are known as tags. The content of an XML element appears between the opening tag (for example <id>) and the closing tag (for example </id>). Closing tags are indicated by a forward slash in front of the tag name.

Most element names can be freely selected, provided the following rules are observed: The first character must be a letter, an underscore, or a colon; the remainder may contain characters or numbers, and the length is essentially unlimited. However, the use of colons should be avoided as they have a different function; and the tag name must not begin with “xml”.

Specification of data interfaces with structured text

XML distinguishes three types of elements: simple elements, complex elements, and empty elements.

Simple elements contain only character strings. In the above example, the order number is a simple element.

<orderno>48729</orderno>

Complex elements contain other elements. In our example, this applies to every position in the order:

<position>

<id>1</id>

<itemno>57823566</itemno>

<quantity>4</quantity>

<priceEUR>15.90</priceEUR>

</position>

Empty elements have no content. The information is the existence of the element and (where applicable) its attributes. The HTML element for line breaks is a typical example:

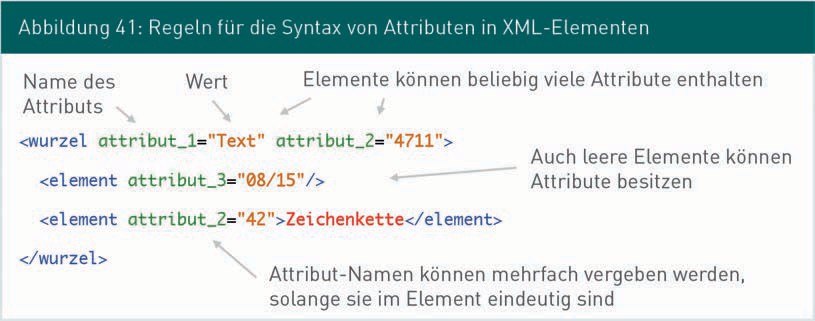
<br />

Attributes are used to assign properties to XML elements. They can only be positioned in the initial tag of the element. The rules for naming attributes are the same as those for tag names. They must also be unique in the element, in other words, they may only occur once in a given element. Attribute values are shown in single or double quotation marks.

Simple, complex, and empty elements The elements in an XML file are divided into simple, complex, and empty elements.

Attributes

An XML element can be specified in greater detail with an unlimited number of different attributes.



Attributes are often a useful way of storing information. They are the smallest unit of information in the XML language and therefore cannot be refined any further with additional information. The use of attributes therefore harbors certain risks in terms of flexibility if data structure changes arise mid-way through operation. Consider the above example: In our example of item ordering, assume that the customer’s name is stored as an attribute of the header data element rather than as a simple element:

<headerdata client="Samuel Berg">

<orderno>48729</orderno>

<order\_dated>2011-12-04</order\_dated>

<customerno>AD-4333 532</customerno>

</headerdata>

If you need the customer’s first name and surname to be listed separately, together with their birth name, the attribute “customer” will need to be removed and replaced with other attributes. These types of changes can be hugely time-consuming. Conversely, by saving the customer as an element, changes can easily be made to the data structure simply by adding more sub-elements:

<headerdata>

<orderno>48729</orderno>

<customer>

<firstname>Samuel</firstname>

<surname>Berg</surname>

<birthname></birthname>

</customer>

<order\_dated>2011-12-04</order\_dated>

<customerno>AD-4333 532</customerno>

</headerdata>

The rules on the structure of XML files are defined using XML schemas which are recorded in a separate XSD (XML Schema Definition) file.

Well-formed An XML file is said to be well-formed if it complies with all general syntax rules.

There are two levels of “correctness” for an XML file: If it complies with all general syntax rules, it is said to be well-formed. In such cases, for example, all tags must have a corresponding end tag, it must only contain one root element and all attributes must be shown in inverted commas. Provided the schema rules specified for the XML file are also observed, this is considered a valid XML file.

Specification of data interfaces with structured text

### Definition of XML languages

XML offers the option of generating and sending a structured text message. As previously mentioned, there must be a clearly defined and verifiable design rule for every XML message. Such rules are defined with the XML language XML schema and stored in a file with the extension XSD (XML Schema Definition).

Note

Valid

An XML file is said to be valid if it is well-formed and complies with all the rules in the specified XML schema.

The XML schema is a highly complex language and we will only touch briefly on it in this module. The W3Schools.com (2014) website is a good reference source with numerous application examples.

###### Example of an XML schema

The following XML schema specifies the XML example of an <order> we considered at the beginning of module 6.2. The individual elements of the XML schema are discussed below.

Note

In XML files and XML schema diagrams, comments begin with <!-- and end with -->.

<?xml version="1.0" encoding="utf-8"?>

<!-- xs:schema is always the root element of an XML schema file -->

<xs:schema [xmlns:xs="http://www.w3.org/2001/XMLSchema"](http://www.w3.org/2001/XMLSchema) elementFormDefault="qualified">

<xs:element name="order">

<xs:complexType>

<xs:sequence>

<xs:element ref="headerdata"/>

<xs:element ref="positiondata"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="headerdata">

<xs:complexType>

<xs:sequence>

<xs:element name="orderno" type="xs:integer"/>

<xs:element name="customer" type="customertype"/>

<xs:element name="order\_dated" type="xs:date"/>

<xs:element name="customerno" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="positiondata">

<xs:complexType>

<xs:sequence>

<xs:element name="position" type="positionType" maxOccurs="unbounded"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:complexType name="positionType">

<xs:sequence>

<xs:element name="id" type="xs:integer"/>

<xs:element name="itemno" type="xs:integer"/>

<xs:element name="quantity" type="xs:integer"/>

<xs:element name="priceEUR" type="xs:decimal"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="clientType">

<xs:sequence>

<xs:element name="title">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="Ms"/>

<xs:enumeration value="Mr"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

<xs:element name="firstname" type="xs:string"/>

<xs:element name="surname" type="xs:string"/>

<xs:element name="birthname" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

</xs:schema>

###### Fundamental principles and general concepts

In terms of its data structure, an XML file represents a tree with precisely one root, and every XML element outside of the root being located in a different element. The leaves of the tree are represented by simple XML elements which do not contain any further elements.

The content of an XML element may therefore be simple (leaf) or complex (child node with further subnodes). Simple elements are of the type simpleType and are used to store simple data types such as int, ﬂoat, string, date, and time. Simple

Specification of data interfaces with structured text

types do not contain any further XML elements. Complex elements are specified as complexType. They contain a defined quantity of simple or complex XML elements.

To specify complex XML elements in detail, first define a separate element type. All admissible XML elements and attributes are defined in the type definition. Next, define the actual name of the XML element and assign the corresponding element type to that element. In this way, the same element types can be applied to different elements.

Element types are defined in a similar way to classes in an object-oriented language. The structure is defined for all instances of that class and is identical in all class objects. If the class is modified, all instances of that class will be modified at the same time. XML elements and XML element types behave in a similar way: If the element type is modified, all XML elements of that type will be modified at the same time.

The XML schema distinguishes between locally and globally defined element types. Local element types are precisely defined in the element corresponding to that type. This type cannot subsequently be reused in other elements. Global element types are declared at the start (or at the end) of the schema file. If an element is structured as a globally declared element type, the type declaration will only specify the reference. These element types are easily and consistently reused multiple times, because the structure of all identical elements is described in exactly one place within XML schema.

###### Simple types

In the above example, the element <title> is a simple element type: <title>Mr</title>. This element may be defined as follows in the XML schema:

<xs:element name="title" type="xs:string"/>

The name of the element is defined with the attribute name and the element type with the attribute type. Table 21 lists some of the typical data types commonly used in XML schemas.

Child nodes

A child node (child element) refers to an XML element that is not a root element. A child element is the “child” of another XML element, i.e. it is contained inside another element.

|  |  |
| --- | --- |
| Table 21: Selection of simple data types used in XML schemas | |
| XSD data types | Description |
| xs:string | Data type for simple character strings without constraints |

|  |  |
| --- | --- |
| XSD data types | Description |
| xs:decimal | Data type for floating-point numbers |
| xs:integer | Data type for whole numbers |
| xs:boolean | Data type for Boolean values (permissible values are: false, true, 0, 1) |
| xs:date | Data type for dates in the format: YYYY-MM-DD; examples: 2014–05–26; 1990–01–23 |
| xs:time | Data type for times in the format: hh:mm:ss; examples: 13:45:33; 09:09:09 |

As a rule, it is possible to limit the value range of simple elements. For example, in the element <title>, the only admissible values should be Ms or Mr. The enumerations explored in Module 5 are a typical application case for limiting values.

The title element is limited to precisely two possible values from the value range of xs:string by means of a restriction. A separate type must be defined for each restriction. The complete definition for the element <title> is as follows:

<xs:element name="title">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:enumeration value="Ms"/>

<xs:enumeration value="Mr"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The element <xs:simpleType> introduces the definition of a simple element type, in this case directly ascribed to the element <title>. The element

<xs:restriction> defines the restriction, in this instance based on the data type xs:string. The two elements <xs:enumeration> define the valid values “Ms” and “Mr” for this restriction. If a value other than Ms or Mr is found in the element <title> in an XML file, the processing systems will automatically identify that XML file as invalid.

Specification of data interfaces with structured text

###### Complex types and composites

In the above example, the element <customer> is an example of a complex element containing four XML elements in total:

<customer>

<title>Mr</title>

<firstname>Samuel</firstname>

<surname>Berg</surname>

<birthname/>

</customer>

Below is an example of an XML schema defining the element <customer> together with all its child elements. For clarity, the element <title> is defined without restrictions:

<xs:element name="customer">

<xs:complexType>

<xs:sequence>

<xs:element name="title" type="xs:string"/>

<xs:element name="firstname" type="xs:string"/>

<xs:element name="surname" type="xs:string"/>

<xs:element name="birthname" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

After defining the element name with the element <xs:element name="customer">, the schema element <xs:complexType> declares the XML element <customer> to be a complex element.

The element <xs:sequence> is a so-called compositor, indicating that the complex element comprises a pre-defined sequence of child elements. Table 22 briefly outlines all the compositors permitted in XML schemas. In an XML schema, the compositors determine the positioning of a complex element’s child elements within the XML files.

|  |  |
| --- | --- |
| Table 22: Overview of XSD compositors | |
| XSD data types | Description |
| <xs:sequence/> | “Sequence” indicates that this group of elements must follow the sequence defined in the schema. Elements declared as optional within <xs:sequence/> may be skipped. |
| <xs:choice/> | “Choice” indicates that precisely one of the elements declared in <xs:choice/> may be contained in an XML file. |
| <xs:all/> | “All” indicates that all elements declared in  <xs:all/> must be contained in the XML file. Their sequence is unimportant. Optional elements are permitted, but elements cannot appear more than once. |

Finally, all child elements of <customer> are declared within the compositor. In our example, these are four simple element types of the type xs:string.

Until now, the type definitions for <customer> have been local, positioned directly on the element definition. For complex elements, type definition is often at a single central point in the XML schema to allow the same element type to be reused for different elements.

Global type definition is achieved by declaring the complex data type, in our example below with the name customerType:

<xs:complexType name="customerType">

<xs:sequence>

<xs:element name="title" type="xs:string"/>

<xs:element name="firstname" type="xs:string"/>

<xs:element name="surname" type="xs:string"/>

<xs:element name="familyname" type="xs:string"/>

</xs:sequence>

</xs:complexType>

The element <customer> is then declared as follows:

<xs:element name="customer" type="customerType"/>

Specification of data interfaces with structured text

The name of the complex type customerType is specified in the attribute type, then the next XML element is declared. The globally defined type customerType can now be reused by all elements assigned the same structure.

Type definitions are written in the schema file, either before or after definition of the root element.

Like globally defined data types, elements can also be reused. For example, the element

<xs:element name="customer" type="customerType"/>

can be reused elsewhere in the XML schema, for example for declaring a <partner> element:

<xs:element name="partner" ref="customer"/>

The attribute ref in the element declaration refers to an element declared elsewhere in the XML schema.

###### Frequencies

The same elements can recur multiple times in XML files. For example, if an order contains multiple positions, the option of specifying all positions in the XML file must be available. There are also optional elements that may be omitted depending on the application case in question. The attributes maxOccurs and minOccurs are used in the XML schema to specify frequencies. The quantity given always refers to frequency within a compositor.

|  |  |
| --- | --- |
| Table 23: Frequencies in the XML schema | |
| Frequencies | Description |
| maxOccurs | maxOccurs determines the maximum frequency with which the element may occur in the compositor. The default setting is 1 and will apply until another value is set. The value “unbounded” allows unlimited incidences of an element.  Within xs:all the mandatory value for maxOccurs = 1. |

|  |  |
| --- | --- |
| Frequencies | Description |
| minOccurs | minOccurs determines the minimum frequency with which the element must occur in the compositor. The default setting is 1 and applies until another value is set. If the value is set at 0, this element is optional.  Within xs:all the mandatory value for minOccurs is 0 or 1. |

In the above example, the element <birthname/> is declared optional, and the attribute minOccurs is therefore used as follows:

<xs:element name="birthname" type="xs:string" minOccurs="0"/>

###### Root element

An XML file must have precisely one root element, but several root elements may be declared within an XML schema file. When generating the XML file, precisely one element is selected from available root elements and the XML file is then built according to its structure. Multiple XML structures within an XML schema may be defined in this way. In practical applications, this allows conceptually related messages which exchange some identical data structures to be combined into a single schema. If complex element types are also globally defined, these definitions will apply to all messages generated on the basis of the schema. In this way, incoming and outgoing messages can be described at a conceptual interface within a file.

### Deriving class diagrams from XML formats

The specification of technical interfaces often necessitates either the integration of existing technical systems via their interfaces or the conversion of internal conceptual data into technical messages between technical systems. In both instances, either XML messages are converted into business objects or vice versa. Both cases can be supported by modeling XML schemas in class diagrams or, conversely, by modeling class diagrams in XML schemas.

Specification of data interfaces with structured text

Below is a simple algorithm used to derive a class diagram from a given XML schema. Figure 42 illustrates the step-by-step process. The sample XML schema from module 6.3 is used here as an example.

###### Step 1

All global elements from the XML schema are modeled as classes in the class diagram.



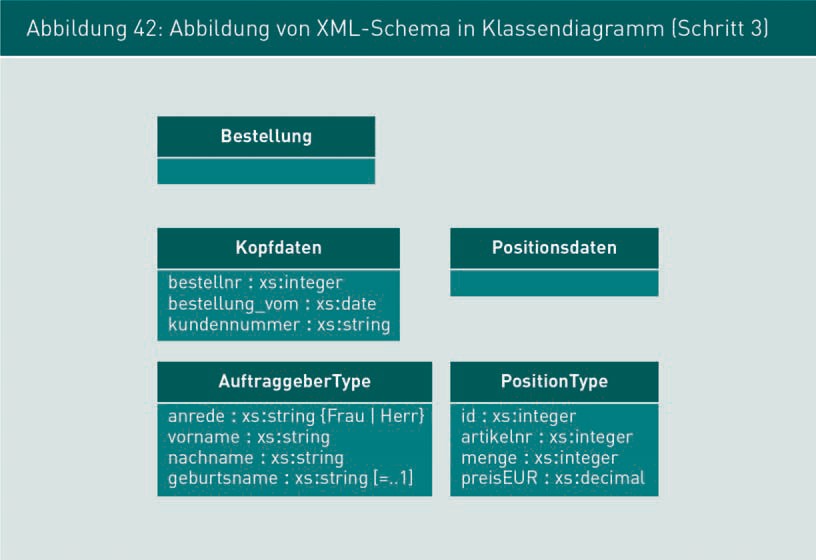
###### Step 2

All global, complex types from the XML schema are modeled as classes in the class diagram.



###### Step 3

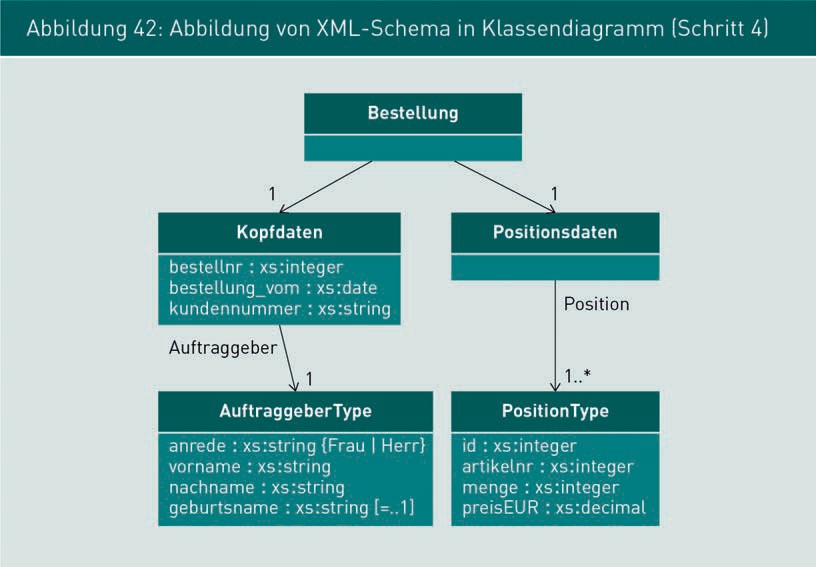
All simple sub-elements of elements and complex types are modeled as attributes in their assigned classes (i.e. the classes derived from their parent elements or complex types). The actual technical data types of the attributes are incorporated to ensure type protection as the project progresses.



###### Step 4

All compositors and multiple incidences of elements are expressed by associations. The direction of the association is derived from the elements’ parent/child dependency. The maxOccurs and minOccurs attributes are also taken into account, even if the default values apply, and denoted as multiplicities in the class diagram.

Specification of data interfaces with structured text



###### Tips and tricks

When modeling XML schemas in a class diagram, you may need to consider the following: The length and other attribute restrictions may be specified as a constraint, for example { length:8 } or { length:2..n }. Simple types with a restricted value range for global use may be modeled as an enumeration class. If the sequence of elements in the XML compositor xs:sequence is important, the attributes can be modeled in the class in their correct sequence if possible. Additionally, the sequence may also be added to the attribute as a constraint, i.e. {order-no=4}.

Modeling the compositor of xs:choice in the class diagram is only possible in certain cases. If the individual elements within xs:choice lend themselves to generalization under an umbrella term, inheritance may be used in the class diagram. If not, they cannot be directly modeled in UML. In such cases, make a note on the diagram stating that only one element from the set may occur.

### Specification of Web services with WSDL

A Web service is a system function provided via a network which can be activated by other systems. The function is usually activated and the response transmitted using structured text, often in the form of XML messages. A description of the functions available as a Web service, their parameters, and return values are likewise usually given as structured text.

A Web service implements a particular function, for example, the current weather and the weather forecast for a particular location or identification of a book’s ISBN number. Web services are characterized by stateless processing. A Web service has no internal state; the same parameters will always return the same result (see module 5.4). A web service also has a unique URL which is used to call it.

The functions and messages of Web services are specified using Web Services Description Language (WSDL). WSDL is an XML language for the specification of Web services. The following elements of a Web service are defined in a WSDL file:

* Functions of the Web service with its input and output messages
* Service address (URL) and
* Other technical information regarding access to the service and deployment.

The following example (see Figure 43) of a Web service can be accessed from the Webservice website (n.d. a). The machine-readable WSDL description is likewise available on the Webservice website (n.d. b). This Web service expects the name and country of a major city as the input value and returns the current weather forecast for that city.



Specification of data interfaces with structured text

###### Structure of a WSDL specification for Web services

We have already examined the concepts of XML and the definition of XML messages. Next, we take a closer look at the fundamental structure of WSDL files using a weather forecast as an example.

It begins with the XML file declaration which is mandatory for all XML files:

<?xml version="1.0" encoding="utf-8"?>

This is followed by the definitions of parameters and their data types, following the sequence prescribed by the WSDL standard.

<wsdl:definitions>

<wsdl:types> […]

The types GetWeather and GetWeatherResponse are illustrated here

<s:element name="GetWeather">

<s:complexType>

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="CityName" type="s:string"/>

<s:element minOccurs="0" maxOccurs="1" name="CountryName" type="s:string"/>

</s:sequence>

</s:complexType>

</s:element>

<s:element name="GetWeatherResponse">

<s:complexType>

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="GetWeatherResult" type="s:string"/>

</s:sequence>

</s:complexType>

</s:element> […]

</wsdl:types>

After the types, the messages are declared. Here is an excerpt from the WSDL with the messages GetWeatherSoapIn and GetWeatherSoapOut, describing an ingoing and an outgoing message. Reference is made to the aforementioned types as the message content:

<wsdl:message name="GetWeatherSoapIn">

<wsdl:part name="parameters" element="tns:GetWeather"/>

</wsdl:message>

<wsdl:message name="GetWeatherSoapOut">

<wsdl:part name="parameters" element="tns:GetWeatherResponse"/>

</wsdl:message> […]

This is followed by a declaration of the functions provided by the Web service. Below is an excerpt for the operation GetWeather, which expects the message GetWeatherSoapIn as the input message and delivers GetWeatherSoapOut as the output message:

<wsdl:operation name="GetWeather">

<wsdl:documentation [xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">](http://schemas.xmlsoap.org/wsdl/) Get weather report for all major cities around the world.

</wsdl:documentation>

<wsdl:input message="tns:GetWeatherSoapIn"/>

<wsdl:output message="tns:GetWeatherSoapOut"/>

</wsdl:operation> […]

Finally, there is a declaration of technical information about the service, including a precise specification of the URL where the service can be accessed on the Internet.

<wsdl:service name="GlobalWeather">

<wsdl:port name="GlobalWeatherSoap" binding="tns:GlobalWeatherSoap">

<soap:address [location="http://www.webservicex.net/globalweather.asmx"/>](http://www.webservicex.net/globalweather.asmx)

</wsdl:port>

</wsdl:service> […]

The definition ends with the closing tag of the root element:

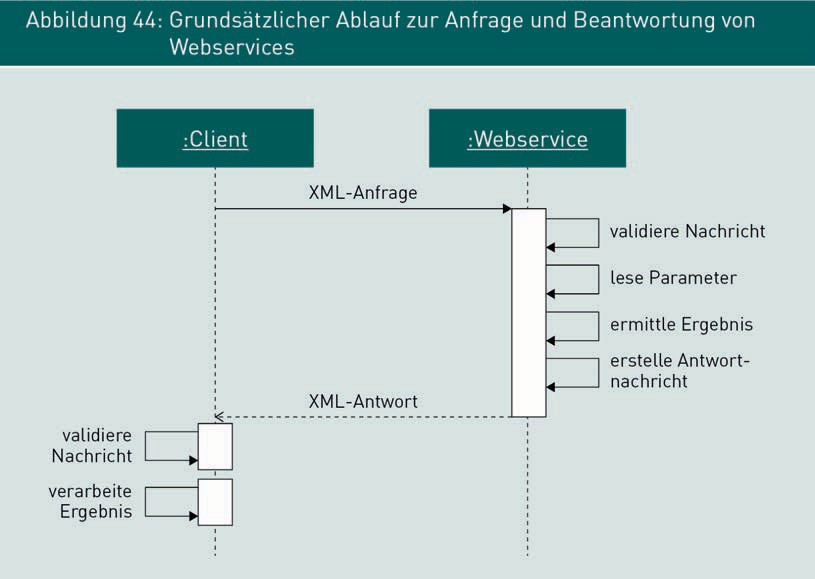
</wsdl:definitions>

The aforementioned elements of the WSDL file for a weather service illustrate the basic structure of such messages. The technical details are too complex to consider in any greater depth here.

###### Basic communication procedure between systems via Web services

The sequence diagram in Figure 4 illustrates the procedure for communication between systems via Web services. In the example shown here, XML messages are being exchanged.

Specification of data interfaces with structured text



The client sends an XML message to the Web service. First, it validates whether the message is well-formed and valid. Next, it reads the parameters and calculates the result on this basis. The result is written into an XML message which is transmitted to the client as a response. The client then checks that the message is well-formed and valid. If the XML message passes both these tests, the result will be processed.

Top tip:

Examples of publicly accessible Web services can be found on the WebserviceX.net and Service Repository websites, among others. The WSDL documents for all services and the functions provided can be retrieved via a Web interface.

Summary

Structured text messages are human-readable messages with a clearly defined and easily verified structure. XML is a common format used in communication between IT systems. XML offers the option of storing information using hierarchically structured but freely selectable elements and their attributes. The rules governing the design of XML messages can be defined using the XML language **XML schema**.

The conversion of XML messages into business objects modeled in a class diagram can be described with an algorithm. This is an easy way to visualize the content and structure of XML messages. Technical interfaces between systems are often implemented as a Web service. In such cases, the functions and messages of the Web services can be specified using the XML language WSDL.



# Unit 7

## Specification of quality characteristics

#### LEARNING OBJECTIVES

After working through this unit, you will know ...

... the properties of quality characteristics and quality models.

... the goal/question/metric method and its various stages.

... how to identify and document quality characteristics that can be verified using the goal/question/metric method.

DL-D-ISPE01-L06

1. Specification of quality characteristics

### Introduction

Until now, we have focused on specifying the structural and behavioral properties of software components and software systems, with a particular emphasis on the specification of functional requirements. Alongside functional requirements, it is also important to specify the quality characteristics of systems and components. This unit will explore models and methods for specifying particular, verifiable quality characteristics.

### Quality and quality models

###### Software quality

In a general sense, ISO standard 9126 defines software quality as follows as follows:

“Software quality is the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.”

Software must therefore be reviewed to determine whether the delivered product meets the previously defined requirements or needs. The standard stresses the importance of specification in this connection, stating that software quality can only be determined on the basis of the specification (“defined requirements”).

This poses the challenge of how to specify quality. The earlier identification of requirements often includes only very generalized quality requirements, if at all. Examples might be: “The system must offer good usability” or “the system must be secure” or “the system must be easy to maintain.” While all these statements concern the system’s quality, they are too general and too abstract for the software architect or developer to derive specific system requirements. Furthermore, these requirements are not suitable for use as the basis for preparing test cases with which to document the required quality. As well as the functional requirements placed on a system or component, it is equally important to specify its quality characteristics.

###### Quality characteristics

Quality characteristics define which qualitative properties the system must fulfill. They do not describe new functions but add qualitative and quantitative characteristics to conceptual functions.

Specification of quality characteristics

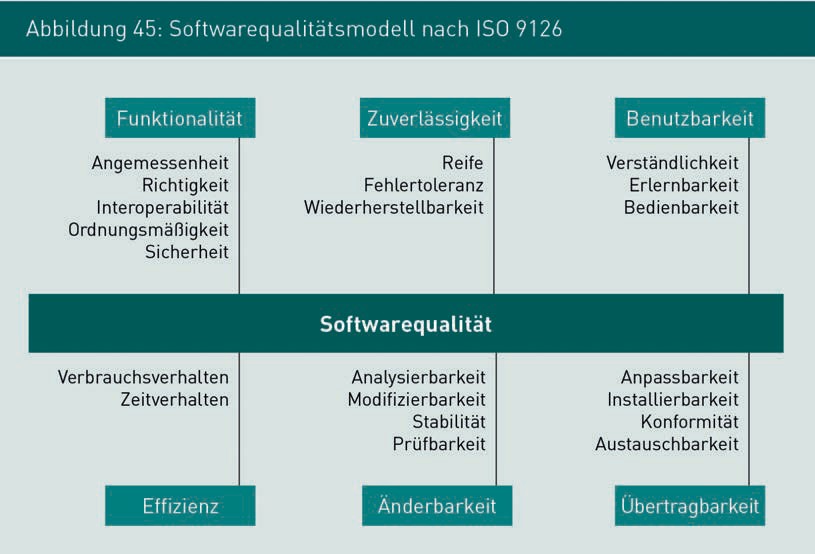
The conceptual/technical specification must refine quality characteristics, firstly to reflect the actual needs of users, and secondly to define the system’s specific, measurable properties.

The totality of quality characteristics ultimately describes a software system’s quality objective. The SMART model used in project management and human resources development is ideal for gauging whether quality characteristics have been adequately described. Both of these industries agree targets and then review their achievement. The definition of quality characteristics for software systems is well-suited to the SMART technique. SMART is an acronym coined by Schiersmann/Thiel (2000) as follows:

* S = Specific: Quality characteristics should be clearly described and defined.
* M = Measurable (qualitative, quantitative): It must be possible to verify implementation of the quality characteristics.
* A = Acceptable: The quality characteristics, including the testing methods used, must be accepted by the project’s stakeholders.
* R = Realistic: Quality characteristics must be achievable, with due regard for functional and organizational framework conditions.
* T = Time-bound: Quality characteristics must be implemented within a set time frame, e.g. in a certain release.

###### Quality model

Quality models, particularly software quality models, are a way of making the abstract concept of “quality” more tangible. The basic precept of a quality model is that the more abstract, general concept of quality is broken down into various sub-concepts. These sub-concepts, in turn, are broken down still further until it becomes possible to formulate a specific, measurable quality characteristic. Quality models are comprised of a hierarchy of concepts used to specify more tangible quality characteristics. Figure 45 shows an example of a quality model to ISO 9126. The term quality is broken down into six sub-features, which in turn are sub-divided into further sub-features.



This type of quality model is also known as an FCM model — factor (in this case, feature), criteria (in this case, sub-feature), and metrics (in this case, indicators) — with reference to the inner structure of the quality model. Table 24 explores the elements of an FCM model in greater detail.

|  |  |  |
| --- | --- | --- |
| Table 24: Elements of an FCM model | | |
| Term used in the FCM model | Meaning | Examples |
| Factor (in this case, feature) | A factor uses a single term to summarize various aspects of quality. It is the first layer of the FCM model. | Reliability, usability, changeability, efficiency, portability, functionality |
| Criteria (in this case, sub-feature) | A criterion focuses on one aspect of a quality feature. If necessary, sub-features may be broken down further into individual aspects (sub-sub-features). | Sub-features of reliability: Maturity, fault tolerance, recoverability |

Specification of quality characteristics

|  |  |  |
| --- | --- | --- |
| Term used in the FCM model | Meaning | Examples |
| Metrics (in this case: indicators) | An indicator is a specific, observable feature of a software system. Indicators allow statements to be made on whether a required sub-feature is fulfilled. | * Duration of system failure until availability is restored * Number of inconsistent data records in the system after misuse by a user * Number of serious system errors when processing defective XML messages |

Other examples of quality models can be found in Eeles (2004) and Boehm (1978), among others.

Organization-specific quality models often exist and describe those features and sub-features which the company considers particularly important. Quality models are just one tool for the structured, methodical specification of quality characteristics.

The following module introduces a technique for identifying specific, verifiable quality characteristics based on a quality model.

### Goal/Question/Metric method (GQM)

The Goal/Question/Metric method (GQM method) is a model for specifying verifiable quality characteristics and performing measurements. GQM was developed by Rombach and Basili in 1987 (Rombach/Basili 1987).

GQM aims to develop quality characteristics which are tailored to a specific software system. The quality requirements placed on the system will vary depending on the nature of the project. The specification of quality characteristics for industrial information systems tends to be very selective and confined to very specific aspects of the system. For example, for systems without a GUI, usability is a far less important consideration than reliability. A small support system for handling internal telephone connections will have very different security requirements than an accounts management system for online banking.

It is not generally possible to specify every conceivable quality requirement for an information system. For example, if a system is comprised of 20 components and we were to specify at least one indicator for each sub-feature of the quality model to ISO 9126 for every component, this would necessitate more than 400 specific quality characteristics.

When specifying quality, therefore, the key is to focus on the most important functions for that particular sector, and to limit yourself to specifying 2-3 quality characteristics initially, only incorporating additional quality characteristics where you feel this is necessary.

The example below illustrates how the GQM method can be used to specify quality characteristics. An online shop has identified user-friendliness as a key quality characteristic. The next step is to ascertain how to specify and measure the user-friendliness of an online shop.

###### Overview of the goal/question/metric method

The GQM method for the specification of quality characteristics can be broken down into the following steps:

* + 1. Define the evaluation goals
    2. Refine the evaluation goals with a quality model
    3. Derive questions
    4. Derive metrics
    5. Define the specific measurements required
    6. Define the measurement procedure and mechanisms
    7. Formulate verifiable quality characteristics

Step 1: Define evaluation goals

The first step when specifying quality is to define specific goals. A goal is formulated using a fixed sentence structure and is sub-divided into the following aspects

* + Purpose: What do I want to achieve? (e. g. improve, monitor, evaluate)
  + Quality characteristic: Which characteristic is under scrutiny here?
  + Object: Which specific element of the system (e. g. component, GUI, interface) is under scrutiny?
  + Perspective: From the perspective of which stakeholder?

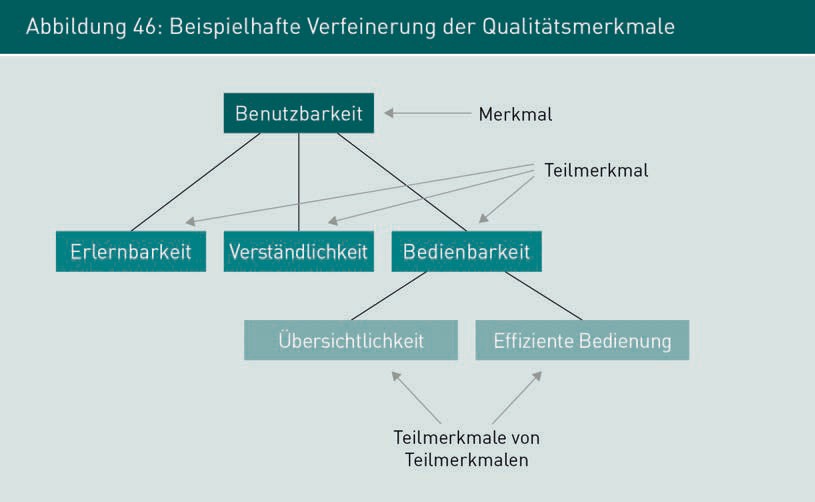
This sharpens the focus for subsequent analysis of the characteristic “user-friendliness”. Table 25 contains a grid for formulating goals and uses two examples to illustrate the elaboration of the question. All subsequent steps in the GQM method are centered around the formulated goals.

Specification of quality characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| Table 25: Grid for formulating GQM evaluation goals | | | |
|  | Explanation | Example 1 | Example 2 |
| Goal | Purpose | Evaluate | Evaluate |
| Quality characteristic | User-friendliness | User-friendliness |
| Object | The ordering GUI | The search function in the product catalog |
| Perspective | From the perspective of a regular customer | From the perspective of a new customer |

Steps 2 and 3: Refine the evaluation goals with a quality model

Despite having sharpened the focus on identified quality characteristics in step one, the formulated goals are usually still too abstract for the derivation of specific metrics, so step 2 refines these quality characteristics further until specific questions can be asked. An example of a quality tree can be seen in Figure 46. Starting from the characteristic “user-friendliness”, the direct sub-feature “usability” is refined for this particular project to include the two sub-features “clarity” and “efficient operation.”



Here, the quality model helps us to derive sub-features as precisely as possible. Identifying and defining the relevant sub-features of sub-features is a creative process. The sub-features are continuously refined until they reach the point where specific questions can be formulated.

In this instance, the stakeholders rate clarity and efficient operation as particularly important. As an example, Table 26 shows the questions formulated in stage 3 of the GQM method. The questions should be formulated in a way which allows compliance with the quality goal to be verified from the answers. As such, the formulated questions should invite yes or no answers wherever possible. Once the questions have been formulated, their connection to the aspired evaluation goal formulated in step 1 should be checked.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 26: Examples of formulated questions in GQM | | | |
|  | Explanation | Example 1 | Example 2 |
| Question | Question to gauge achievement of the quality goal (ideally with a yes or no answer). | Efficient operation: Is the ordering process quick? | Clarity: Are the search results clearly and accessibly displayed? |

Steps 4 and 5: Derive indicators and specify the metrics

Step 4 should define the indicator for answering the question. The first step is to define the yardstick or metric used for measuring, independently from the actual measurements. In other words, the “unit” of measurement.

As a general rule, both objective and subjective metrics are supported. Objective metrics can be measured directly on the system. Examples include the lines of code (LOC) in the program code, the proportion of automated test cases, or the system response times. Subjective metrics are obtained from surveys or tests with individuals who work on or with the system. Examples include the empirical outcome of a user survey, a survey of the support team to gauge the intelligibility of the screen masks, or a survey of participating developers to gauge the system’s ease of maintenance.

Some examples of metrics for the above evaluation goals can be found in Table 27. When formulating the indicators, a target measurement is not yet defined at this stage.

Specification of quality characteristics

|  |  |  |  |
| --- | --- | --- | --- |
| Table 27: Examples of objective and subjective measurements in GQM | | | |
|  | Explanation | Example 1 | Example 2 |
| Metric | Unit(s) or parameter(s) used for measurement purposes to answer the question | Number of dialog boxes that appear from the shopping cart through to finalization of the order | Assessment of clarity on a scale of 1 (very clear) to 5 (very unclear) |

The target values are defined in step 5; see Table 28 for an example. The values defined here provide the development team with guidance when designing the system. Later when the system undergoes quality assurance testing, these values will be checked.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 28: Examples of target measurements in GQM. | | | |
|  | Explanation | Example 1 | Example 2 |
| Quantity | Define acceptable measurements for the indicators | 2 pages for  regular customers, 4 pages for new customers | Average score of 2.0 |

The definition of target measurements is essentially a quantification of quality characteristics. In other words, the defined system quality is determined from the totality of all defined target values.

Step 6: Define the measurement procedure and mechanisms

As well as determining the metrics for measuring values, it is also necessary to specify how the measurement should be conducted. With subjective metrics in particular, the quality of the description of measurement is largely determined by the technique employed. For example, the outcome of a survey of 10 members of the development team on the GUI’s user-friendliness would be evaluated differently to professional telephone interviews with 100 randomly selected individuals from the online shop’s target audience. Table 29 describes the calculation methods defined for our sample goals.

|  |  |  |  |
| --- | --- | --- | --- |
| Table 29: Examples of measurement techniques in GQM | | | |
|  | Explanation | Example 1 | Example 2 |
| Measure | Specify the measurement method used | Count the dialog boxes with test orders | Online survey of at least 100 people, including at least 30 potential customers |

Specifying the measurement technique simultaneously serves to gauge the testability of the quality characteristic. If the measurement cannot be calculated, or if a reliable calculation cannot be made for technical or financial reasons, steps 4 and 5 should be repeated until a feasible measurement technique has been identified.

Step 7: Formulate verifiable quality characteristics

Based on the questions and values identified in steps 1 to 6, it is now possible to specify precise, verifiable quality characteristics. To this end, the results from steps 1 to 6 are reformulated into a requirement incorporating the metrics, the object to be measured, and the target measurements. With subjective metrics, the measurement technique should likewise be taken into account.

As an example, these are the requirements identified for usability of the online shop:

Example 1

When ordering, the number of dialog boxes in the GUI from the shopping basket through to finalization of the order should not exceed a value of 2 for existing customers and 4 for new customers.

Example 2

An online survey of at least 100 people including at least 30 potential customers should be conducted to evaluate the clarity of the search function in the product catalog, rating it on a scale of 1 (very clear) to 5 (very unclear), and should achieve an average score of 2.0.

Summary

During the early stages of identifying requirements, usually only very general quality requirements are documented, if at all. However, alongside its functional requirements, the quality characteristics of a system or component must also be precisely defined in a verifiable format (also: SMART).

Specification of quality characteristics

Quality models help to concretize quality by breaking down the more abstract, general concept of “quality” into a series of sub-concepts.

The GQM method is a quality process model for the specification of verifiable quality characteristics in a given project. Questions, metrics and target values are developed to help formulate specific quality characteristics.

146 Annex 1

Bibliography

Apple Mac OSX UI Element Guidelines (n.d.). (URL: https://developer.apple.com/ library/mac/documentation/userexperience/conceptual/applehiguidelines/Controls/ Controls.html#//apple\_ref/doc/uid/TP30000359-TP6 [last accessed: 08.05.2014]).

Assmann, D./Kalmar, R./Punter, T (2002): *Messen und Bewerten von Webapplikationen mit der Goal/Question/Metric Methode.* Handbuch, IESE-Report, Fraunhofer IESE 087.02/D, p. 11–32. (URL: <http://publica.fraunhofer.de/documents/N-17642.html> [last accessed: 24.07.2014]).

Balzert, H. (2009): *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering.* 3rd edition, Spektrum Akademischer Verlag.

Balzert, H. (2011): *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb.* 3rd edition, Spektrum Akademischer Verlag.

Balzert, H. (2008): *Lehrbuch der Softwaretechnik: Softwaremanagement.* 2nd edition, Spektrum Akademischer Verlag.

Boehm, B. W. (1978): *Characteristics of Software Quality.* North-Holland Pub. Co.

Christensen, E./Curbera, F./Meredith, G./Weerawarana, S. (2001): *Web Services Description Language (WSDL) 1.1, W3C.* (URL: <http://www.w3.org/TR/wsdl> [last accessed: 24.06.2014]).

Eeles, P. (2004): *Appendix B: Architectural Requirements, IBM.* (URL: http:// [www.ibm.com/developerworks/rational/library/4708.html](http://www.ibm.com/developerworks/rational/library/4708.html) [last accessed: 24.06.2014]).

Evans, E. J. (2003): *Domain-Driven Design: Tackling Complexity in the Heart of Software.*

1. Edition. Addison Wesley.

GOV (n.d.): *Excise Movement and Control System: How To Register And Use.* (URL: http:// [www.hmrc.gov.uk/emcs/uis-launch-emcs-service.pdf](http://www.hmrc.gov.uk/emcs/uis-launch-emcs-service.pdf) [last accessed: 08.05.2014]).

Java Server Faces (n.d.). (URL: <http://www.exadel.com/tutorial/jsf/jsftags-guide.html> [last accessed: 08.05.2014]).

Kiltz, F. (2010): *Java Webservices.* 1st edition, mitp.

Microsoft User Interface Element Reference (n.d.). (URL: http:// sdn.microsoft.com/en-us/library/windows/desktop/dd373675(v=vs.85).aspx [last accessed: 08.05.2014]).

Object Management Group (OMG) (2014): *Uniﬁed Modeling Language (Uml), V2.4.1.* (URL: <http://www.omg.org/spec/UML/2.4.1>[last accessed: 24.06.2014]).

##### Annex 1

147

Bibliography

RichFaces (n.d.). (URL: <http://showcase.richfaces.org/richfaces/component-sample.jsf> [last accessed: 08.05.2014]).

Rombach, H. D./Basili, V. (1987): *Quantitative Software Qualitätssicherung: Eine Methode für Deﬁnition und Nutzung geeigneter Maße.* In: German Computer Society, GI Informatik Spektrum, June 1987.

Rupp, C./Queins, S. (2012): *UML 2 glasklar: Praxiswissen für die UML-Modellierung.*

4th edition, Carl Hanser Verlag.

Schiersmann, C./Thiel, H.-U. (2000): *Projektmanagement als organisationales Lernen.*

1st edition, Leske + Budrich.

Sommerville, I. (2007): *Software Engineering.* 9th edition, Pearson.

Vonhoegen, H. (2013): *Einstieg in XML: Grundlagen, Praxis, Referenz.* 7th edition, Galileo Computing.

W3Schools.com (2014): *XML Tutorial.* (URL: <http://www.w3schools.com/xml> [last accessed: 24.06.2014]).

W3Schools.com (2014): *XML Schema Tutorial.* (URL: https://[www.w3schools.com/xml/](http://www.w3schools.com/xml/) schema\_intro.asp [last accessed: 24.06.2014]).

Webservice (n.d. a): *Global Weather.* (URL: <http://www.webservicex.net/globalwea-> ther.asmx [last accessed: 24.06.2014]).

Webservice (n.d. b). (URL: <http://www.webservicex.net/globalweather.asmx?WSDL>[last accessed: 24.06.2014]).