

UML-Glossary

Preamble

The content of this document has been a part of the handbook *Project Management with UML and Enterprise Architect* for Version 9, ISBN-13:978-3-9502692-1-5. Releasing Enterprise Architect 11, this book has been revised completely. To keep the new version compact, we decided to release the UML-Glossary as pdf-download.

We also decided to change the title: Compendium of Enterprise Architect from SparxSystems, ISBN 978-3-9503784-1-2. The German version Kompendium zu Enterprise Architect von SparxSystems, ISBN 13 978-3-9503784-0-5 is also available at SparxSystems.

The new books are still intended as procedure documentation for the training [UML with Enterprise Architect](#), but may be used for self-study too.

The handbooks have been extended with further chapters:

- Team Collaboration – multiple users for one model
- Transparent versioning for service oriented architectures
- Colors of Enterprise Architect
- Element Discussions
- Model Mail
- Comprehensive Documentation: optimized Model-Structure

Ordering information and information about actual versions of the handbook can be found at [SparxSystems](#).

For the sake of readability were omitted gender-neutral language. Of course the information and explanations addressed in this book to people of both sexes.

This documentation has been compiled and checked with great care. Unfortunately, however, it cannot be assumed that errors herein do not exist. The author therefore assumes no responsibility or liability for inaccurate entries. The included screenshots have been taken from Enterprise Architect 11.1, build 1113 in most instances. Using other builds, your screens may be different.

Copyright

© 2014 Sparxsystems Software GmbH Vienna. All rights reserved. No part of this document may be electronically modified, published or distributed without the express written permission of the publisher, SparxSystems Software GmbH. The content is exclusively available for on-line access or download for read-only purpose.



The Authors



Ing. Dietmar Steinpichler is a qualified engineer who operated his own real-time systems development company. His previous engagement was for a telecom as business analyst and designer. His key competencies are programming language development in CTI, pattern recognition and abstraction algorithms. As technical project leader, his team handled many major projects with UML modeling tools and distributed architecture.

Since 2007, Mr. Steinpichler acts across Europe as trainer and consultant for Sparxsystems Software GmbH with focus on quality assurance, project processes and requirements management.

Email: dietmar.steinpichler@sparxsystems.eu



Dr. Horst Kargl is engaged in object oriented modeling and programming since 1998. Before joining SparxSystems he was a teaching scientific assistant at the Technical University of Vienna, involved in several research projects with focus on e-learning, semantic web and model driven software development. His study for a Phd was concerned with automatic integration of modeling languages.

Already acting as a freelancer for SparxSystems during his study, he joined SparxSystems Europe in September 2008 as an employee, focused on software architecture, code generation and customization of Enterprise Architect.

Email: horst.kargl@sparxsystems.eu

Contents

UML-Glossary.....	1
Preamble.....	1
Copyright.....	1
The Authors.....	2
Contents.....	3
Introduction to UML.....	5
Documentation.....	5
Advantages of UML.....	5
UML Standard.....	5
UML Extensions in Enterprise Architect.....	6
Historical Development of UML.....	6
Diagram Implementation.....	9
Fundamentals of Behavioral Modeling.....	10
Use Case Diagram.....	11
Actors.....	11
Use Case.....	12
System (System Boundary).....	12
Relationships.....	12
Use Case Relationships.....	13
Descriptions and Notes.....	16
Graphical Elements.....	16
Example.....	17
Chapter Review.....	18
Activity Diagram.....	19
Activities.....	19
The Token-Concept for Activity-Diagrams.....	19
Connections.....	20
Junctions (Decision and Fork=Parallelization).....	21
Merge.....	21
Synchronization (Join).....	22
Composition of Activity Diagrams.....	22
Responsibility Zones (Swimlanes).....	23
Asynchronous Processes.....	24
Interrupt Region.....	24
Graphical Elements.....	25
Example.....	28
Chapter Review.....	30
State Machine Diagram.....	31
States.....	32
Transitions.....	32
Symbols.....	32
Example.....	33
Chapter Review.....	34
Class Diagram.....	35
Class.....	35
Object.....	36
Attributes.....	37
Methods (Operations).....	37
Relationships.....	37
Interfaces.....	43
Symbols.....	46
Example.....	47

Chapter Review	48
Package Diagram	49
Chapter Review	51
Interaction Diagrams	52
Sequence Diagram.....	52
ExecutionOccurence	52
Message Types	52
Symbols.....	54
Example	54
Chapter Review	56
Communication Diagram	57
Symbols.....	58
Example	58
Sequence Diagrams vs. Communication Diagrams	59
Chapter Review	60
Interaction Overview Diagram	61
Component Diagram	62
Symbols.....	62
Example	63
Deployment Diagram	64
Symbols.....	64
Example	65
Chapter Review	66
Timing Diagram.....	67
Composite Structure Diagram.....	67
Object Diagram	68
Chapter Review	69
Images	70
Recommended Additional Literature	72

Introduction to UML

UML is a standardized graphical display format for the *visualization, specification, design and documentation* of (software) systems. It offers a set of standardized diagram types with which complex data, processes and systems can easily be arranged in a clear, intuitive manner.

UML is neither a procedure nor a process; rather, it provides a “dictionary” of symbols – each of which has a specific meaning. It offers diagram types for object-oriented analysis, design and programming, thereby ensuring a seamless transition from requirements placed on a system to final implementation. Structure and system behavior are likewise shown, thereby offering clear reference points for solution optimization.

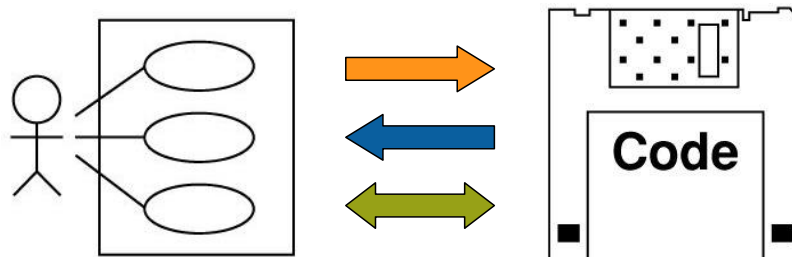


Fig. 1: Forward, Reverse and Round-Trip Engineering

Documentation

One major aspect of UML is the ability to use diagrams as a part of project documentation. These can be utilized in various ways in the most diverse kinds of documents; for example, Use Case Diagrams used in describing functional requirements can be specified in the requirements definition. Classes or component diagrams can be used as software architecture in a design document. As a matter of principle, UML diagrams can be used in practically any technical documentation (e.g. test plans) while also serving as part of the user handbook.

Advantages of UML

The use of UML as a “common language” leads to an improvement in cooperation between technical and non-technical competencies like project leaders, business analysts, software/hardware architects, designers and developers. It helps in the better understanding of systems, in revealing simplification and/or recoverability options, and in the easier recognition of possible risks. Through early detection of errors in the analysis and design phase of a project, costs can be reduced during the implementation phase. The advantages associated with Round-Trip Engineering offer developers the ability to save a great deal of time.

Although UML was initially developed for the modeling of software systems, it can also be implemented for any hardware or organizational project. By allowing processes to be visualized, they can subsequently be analysed and improved. Developers of embedded systems or real-time systems may use non-object-oriented programming languages – applying UML makes sense in this case also!

UML Standard

The official specification for UML 2.4.1 is a complex work of over one thousand pages (<http://uml.org>), arranged into the following sub-components:

- Infrastructure (Architectural Core, Profiles, Stereotypes),
- Superstructure (Static and Dynamic Modeling Elements),
- OCL (Object Constraint Language) and
- Diagram Interchange (UML Exchange Format)

The present book covers only the most important UML core elements and in no way constitutes a complete and comprehensive source of reference. For additional and more specific details on UML, more advanced literature is referred to (see attachment).

UML Extensions in Enterprise Architect

Enterprise Architect uses the extension mechanism (Profile) designated in UML to provide new elements – such as an element for Requirement – as well as additional diagram types. Likewise, extended properties – such as test windows, job orders, risks, etc. – can also be prepared. Thereby, a UML-based tool emerges which, together with a likewise integratable development platform, enables comprehensive project work including requirements management, operational documentation and more.

Historical Development of UML

Despite the fact that the idea of object orientation is more than 30 years old, and the development of object-oriented programming languages spans almost the same length of time, the first books on object-oriented analysis and design methods didn't appear until the early 1990's. The godfathers of this idea were Grady Booch, Ivar Jacobson and James Rumbaugh. Each of these three “veterans” had developed his own method, each one specialized in and limited to its own area of application.

In 1995 Booch und Rumbaugh began to merge their methods into a common Unified Method (UM) notation. The Unified Method was soon renamed as Unified Modeling Language (UML), a more adequate term since it is mostly concerned with the unification of the graphical presentation and semantics of modeling elements, and does not describe an actual method. Indeed, “modeling language” is basically just another term for notation.

A short time later Ivar Jacobson joined in the foray and his Use Cases were soon integrated. From that point on, these three pioneers called themselves the “Amigos”.

Although the methods of Booch, Rumbaugh and Jacobson were already very popular and held a large market share, the Unified Modeling Language (UML) became a quasi-standard. Finally, in 1997, UML Version 1.1 was submitted to the Object Management Group (OMG) for standardization, and accepted. The versions 1.2 to 1.5 both contain several important corrections. In 2004, Version 2.0 was approved with many key modifications and extensions as the new standard. UML 2.3 has become standard in 2010, current version UML 2.4.1 has been published in August 2011 (Source: OOSE).

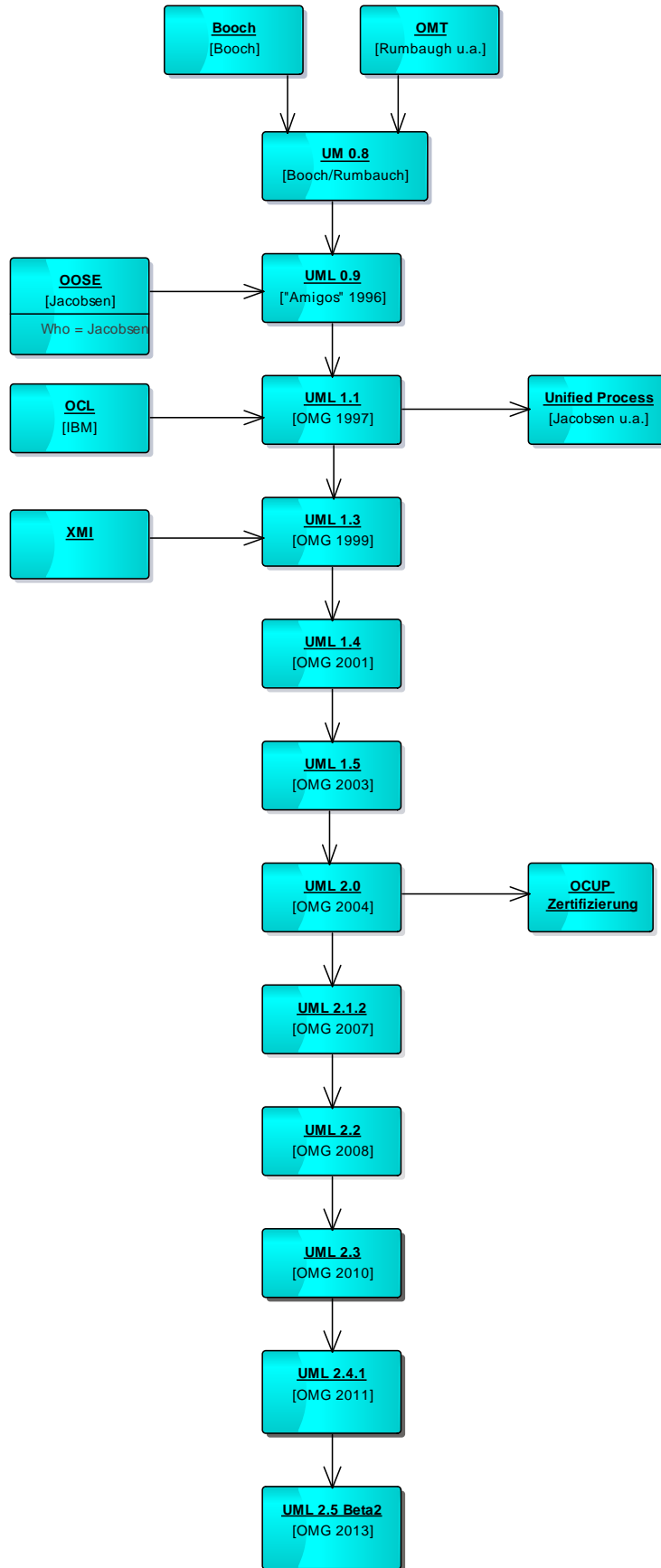


Fig. 2: Historical Development of UML

UML Diagram Types

Officially, in UML there is no diagram overview or categorization. Although UML models and the Repository behind the diagrams are defined in UML, diagrams - or special Repository views - can be relatively freely specified.

In UML, a diagram is actually more of a collection of notation elements. It is in this way, for example, that the package diagram describes the package symbol, the Merge relationship, etc. A Class diagram describes classes, associations, etc. Naturally, however, classes and packages may still be displayed together in one diagram.

A diagram is composed of a diagram space enclosed by a rectangle with a diagram header in the upper left corner. This diagram header shows the diagram type (optional), diagram name (obligatory) and parameter (optional).

The diagram type is, for example, *sd* for Sequence Diagram, or *cd* for Class Diagram. The Parameter field is important for customizable models.

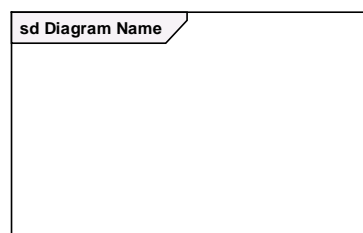


Fig. 3: Diagram Frame Example

UML Version 2.4 contains 13 diagram types which can be divided into roughly two groups. The group structure diagrams represent the static aspects of a system, while the group of behavioral diagrams represent dynamic components.

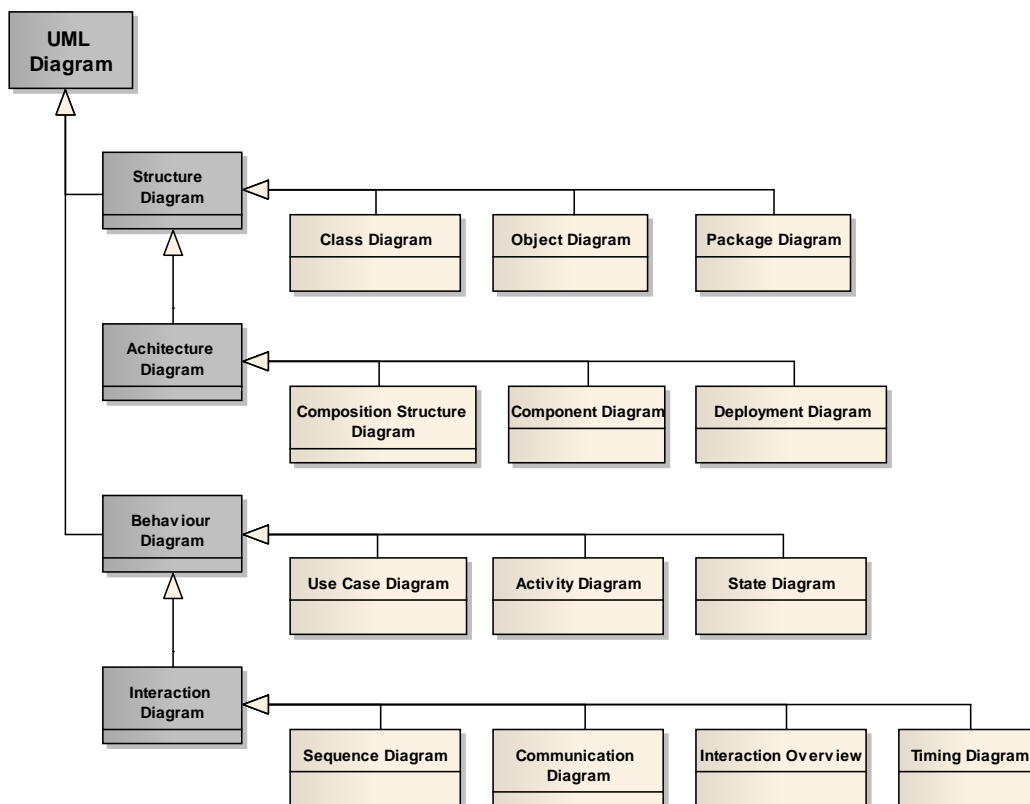


Fig. 4: Overview of UML Diagrams

Diagram Implementation

To many who are new to UML, the question soon arises as to how these diagrams are actually associated with one another. This is a very legitimate question; however, UML itself does not give us a clear answer. It is primarily the software development methodology, or rather the background processes thereof, which can best answer this question. One possible approach concerning the order, or phases of a project, in which the diagrams can be implemented can be shown as follows:

Use Case Diagram	Analysis Phase
<ul style="list-style-type: none"> ○ which use cases are included in the application to be generated ○ which actors are released by this use case ○ what use case dependencies are interconnected, e.g. <ul style="list-style-type: none"> ○ whether one use case is contained within another ○ whether one use case represents a specialization of another ○ whether one existing use case is extended by a second 	
Activity Diagram	Analysis & Design Phase
<ul style="list-style-type: none"> ○ which steps will be taken within a use case ○ what state transition the involved objects experience when handling changes from one activity to another 	
Package Diagram	Analysis & Design Phase
<ul style="list-style-type: none"> ○ into which packages the application can be subdivided ○ which packages allow further subdivision ○ what levels of communication must be realized between packages 	
Class Diagram	Analysis & Design Phase
<ul style="list-style-type: none"> ○ which relations have to be obtained within a project definition (domain model) ○ which classes, components and packages are involved ○ via what type of communication cooperation is to take place ○ which methods and properties do the classes require ○ what are the minimum and maximum numbers of objects to be linked ○ which classes are responsible for multiple objects as a container 	
Sequence Diagram	Design Phase
<ul style="list-style-type: none"> ○ which methods are responsible for communication between selected objects ○ how the chronological cycle of method calls between selected objects is to occur ○ which objects in a sequence are to be newly created or destroyed 	
Communication Diagram	Design Phase
<ul style="list-style-type: none"> ○ how selected objects communicate with one another ○ in which order the method calls are carried out ○ which alternative method calls exist should they be required 	
State Diagram	Design Phase
<ul style="list-style-type: none"> ○ what state transitions are released by which method call ○ which condition will be assumed following object creation ○ which methods destroy the object 	
Component Diagram	Design Phase
<ul style="list-style-type: none"> ○ how are soft and/or hardware parts capsuled with defined function and defined interfaces ○ which components have interfaces to one another ○ what software parts create functionality in components 	
Deployment Diagram	Design Phase
<ul style="list-style-type: none"> ○ which PCs in the application work together ○ what application module will be run on which PC ○ on which communication options should cooperation be based 	

If necessary, the order of diagram use may deviate from that shown in the table because, for example, the division of the work of multiple programmers cannot be managed. In such case, the package diagram can first be created with the class diagram. This order serves to show only one possibility of how you can realize a model of your application, and how the phase transitions can be formulated.

The area of application will also have an effect; the resulting diagram order of implemented diagrams for a business automation job will differ considerably from that of an embedded or real-time task job.

Fundamentals of Behavioral Modeling

Modeling of Behaviors concerns the description of processes, chronological dependencies, state changes, the treatment of events, and the like. UML is object-oriented, therefore behavior is nothing which exists independently, but rather always affects certain objects. When examined in detail, the execution of a Behavior can always be traced to an object.

Every behavior results from actions of at least one object, and leads to changes in the states of the involved objects.

In UML, behavior is principally event-oriented. The execution of behaviors is always triggered by an event. Two special events always occur: the Start event and the End event.

Behavior can be started either directly (i.e. CallBehaviorEvent) or indirectly via a Trigger (TriggerEvent), such as when a point in time is reached (TimeEvent), when a message is received (ReceivingEvent), or when a particular value is reached or changed (ChangeEvent).

In UML, four different specifications are provided for behavior descriptions:

- State diagrams (state machines)
- Activities and actions (activities)
- Interactions (interaction)
- Use cases (use cases)

A use case diagram is actually a structure diagram, since the use case diagram itself doesn't describe processes and behavioral patterns, but rather only the structure (relationships) of use cases and actors. Nevertheless, the use case diagram is categorized as a behavioral diagram in many UML publications. The content of this diagram type is showing the desired functionality, the desired outcome – is the argument.

Use Case Diagram

Use Case diagrams provide a very good overview of the entire system on a highly abstract level. They describe functionality - services and activities to be performed - from the view of the user, and act as the interfaces to the environment. It is important to consider that Use Case diagrams themselves cannot describe behaviors and processes, but rather only the associations between a number of use cases and the involved actors. These may be used for the analysis and management of requirements. Likewise, no order of appearance of described activities/services is shown. A major advantage of the Use Case diagram lies in the structuring of task assignment; subsequently - what will be delivered by the system? - all further specifications can be hierarchically ordered and extended below as Sub Use Cases or other model parts. Help in securing projects by quickly determining job scope and evaluating cost is another advantage. Use Cases provide an overview of function on top of the documentation of the planned system.

This type of diagram describes the goals of the user and is especially good for analyzing the functional requirements placed on a system. It is comprised of only a few yet very intuitive elements and, due to its simplicity, is very well suited for communication between principal (customer) and agent (sub-contractor). Both parties can develop a common view of the system, thereby helping to avoid misunderstandings concerning operational scope in a timely manner.

A Use Case diagram is the graphical representation of the Use Cases and their relation with the environment (interacting users) and their relations with each other. Important information is stored within the textual meta-content of the Use Cases or within diagrams behind, providing detailed information for each one. Use Cases may contain: A self-explaining name, an explanation of the name (note), pre- and post-constraints and a scenario describing the necessary steps to fulfill the functionality/service.

By collecting all important functional requirements by Use Cases it's also possible to plan and discuss all necessary acceptance test cases; test cases for the functionality, for each constraint, for the assigned non-functional requirements and for the included scenarios.

This Use Case diagram shows two use cases and their associated actors. When read from top to bottom, these two use cases suggest a particular order, but in UML this is neither given nor intended. The diagram merely describes what use cases there are and the involved parties. Process flow and order can be described textually within the scenario or later on within additional behavioral models by using activity, state or sequence diagrams.

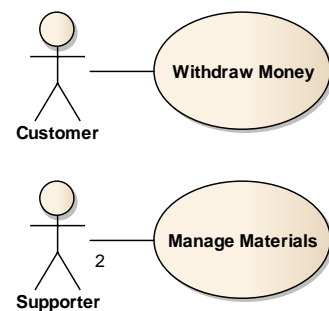


Fig. 5: Use Case Diagram

Actors

In a use case diagram, all parties (stakeholders) involved in a procedure are portrayed with the help of Actors. An Actor is defined as a role outside of the corresponding use case system, and which interacts with the system within the context of the use case. Actors can be persons who use the system or external systems which access the system or interact with the system. They have demands and interests on the system, and are accordingly interested in the results. There can also be events which are triggered without the involved parties (e.g. time events).

An Actor describes a role, which may be replaced by a discrete person (or system) when realized. For example the role "Customer" can be replaced by any person being a *customer of the bank*. In special cases, when the role cannot be replaced by a discrete person (or system), it has to be marked as *Abstract* to express that the actor is a generic role. To qualify an UML element as an abstract element, the name of the element will be shown in *italic* in an UML diagram.

Stereotypes may be used to categorize actors like sensors, timers, actuators, environmental influence, etc.

Notations for Actors

The following illustration shows different notations of an actor. UML provides the stick figure as the Actor symbol. The role name of the actor is placed above or below the figure. It is possible to use any user-specific symbol. The block located to the right is the *node* symbol from the deployment diagram type. The use of a block-type symbol (or similar) to indicate an external system¹ is widespread, as a stick figure usually indicates a human user.

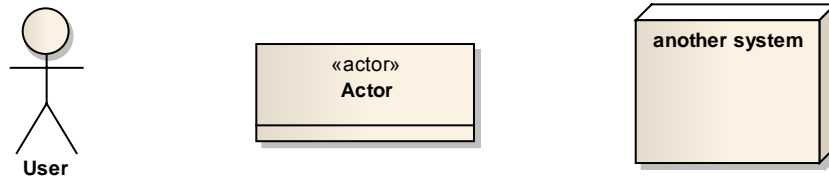


Fig. 6: Notation of Actors

Use Case

A use case specifies a number of actions executed by a system and which lead to a result which is usually important to an actor or stakeholder. Use cases are the activities which one names when describing a process. As an example: for a ticket system, that would be the buying, reserving or cancelling of tickets.

The following illustration shows various forms of notation of use cases. The illustration on the left is standard notation. It is also possible to note the names of the use cases under the ellipse. The advantage here is that the size of the ellipse must no longer scale to the name of the use case. Interestingly, unlike actor notation, placing the name above the ellipse for use cases is not practiced in UML.

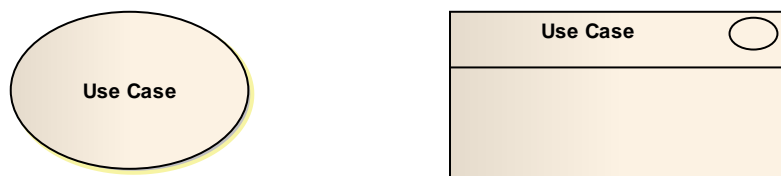


Fig. 7: Notation of use cases

System (System Boundary)

System is not a strictly UML modeling element. System can be understood as the context of the use case in which the use cases of specific actions are executed. System can be a class or a component which represents the entire application. The system is represented by one or more system frames (boundaries); Use cases – services and activities - to be performed by the system are shown in the system frame.

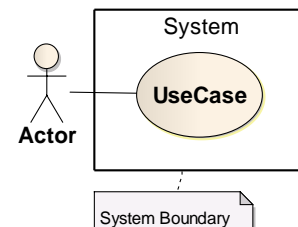


Fig. 8 System

Attention: To draw actors within a boundary will be incorrect!

Relationships

Use cases and actors have a certain relationship with one another. Relationships are modeled with lines. Linking actors and use cases in this way means that both communicate with each other. An actor is linked to use cases using simple association. This indicates an interaction with

¹ right click on element in the diagram | Advanced | Use Rectangle Notation

the system belonging to the use case, and in the context of that use case. Relations can carry additional information, if needed.

It's possible to add multiplicity²; a multiplicity on the use case side indicates how often this use case can be executed by this actor at the same time. Without a writing, multiplicity will be 0..1 by default. On actors side a written multiplicity means the number of actors of the given role to be involved when interacting with the use case. Without a writing, multiplicity on actors side will be 1..1, which can be written as 1 with same meaning.

It is not common for one to use navigable relations; however, these are allowed. These do not represent a specific direction of data flow – as they are usually interpreted – but rather indicate the initiator of the communication between actor and system.

Indicating a navigable association (arrow to or from an Actor), even more semantics with a use case can be expressed. A directed association describes which part is the active and which is the passive. When an actor navigates to a use case, then the Actor is the active party and initiates the use case. Vice versa, in navigation from use case to actor, the actor is passive and will be required and requested by the use case to participate.

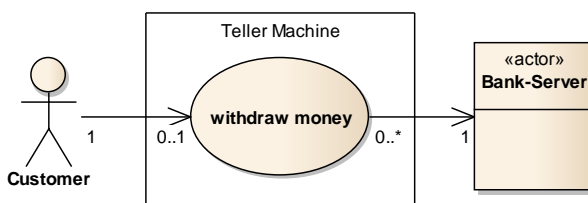


Fig. 9 Multiplicity and active/passive actors

The left sided example Fig. 11 expresses that the *Customer* triggers the use case *withdraw money*, but once a time. To process *withdraw money* the *Bank-Server* is needed (it's passive). While the *Bank-Server* can be involved in any number of *withdraw money* use cases at the same time, the *Customer* can be involved only once at the same time.

Use Case Relationships

Use cases can also be reliant on one another

- With the “Include” relationship, a use case is bundled into and is a logical part of another use case. It represents a compulsory relationship and is therefore often referred to as “*must-relationship*”.
- With the “Extend” relationship, however, one can also specify that a use case is to be extended under certain conditions and on another specific point (the so-called extension point). It represents an optional relationship and is therefore often referred to as “*can-relationship*”.
- Use cases can also be generalized, whereby the general rules apply. Use cases can also be abstract (italics) and first be made clear via sub-use-cases (specialized Use Cases).

Include Relationship (Include)

A part of a use case which appears in the same identical form in other use cases may be transferred to its own use case and re-integrated universally via an Include relationship in order to avoid the redundant specification of these identical parts. Unlike the Generalization relationship, when utilizing the Include relationship no characteristics are passed on.

The Include relationship is illustrated by a dashed arrow with open point in the direction of the use case to be included. The key word «include» is noted for the arrow. An actor must not necessarily be linked to the integrated use case.

Integrated use cases are often provided with the stereotype “secondary”. This is not UML-Standard, but it is in common use since they are normal incomplete (use case fragments) and must be distinguished from the primary (normal) use cases.

² The multiplicity is a time-dependent value with a lower and an upper border, written as x..y, indicating how many instances of the element are needed. Multiplicity describes the amount of possible instances, cardinality on the other hand a concrete amount.

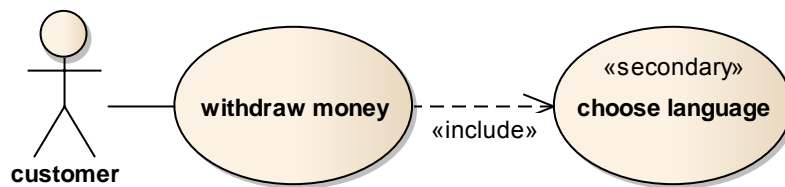


Fig. 10: Example of «include» relationship

Within a use case diagram an include relation specifies that the use case always uses the second one. The timing of the usage is not expressed by the diagram itself, it may be described within the *use case scenario* or by a behavioral diagram describing this use case in detail.

Hint: Please pay attention to *include* only use cases of the same specification level.

Functionality needed several times should be modeled as use case once and can be used by others with relations any time needed. Included use cases are always processed when the use case pointing to it with «include» is executed.

Extend Relationship

If a part of the tasks are transferred from one circumstance to another, this is modeled in its own use case. The arrow is given the stereotype “extend”. The Extend relationship points to the use case to be extended, and starts from that use case which describes the extension's behavior.

This has been defined by the inventors of UML, they preferred to have “extend” instead of “extended by”.

A use case can define any number of extension points optionally. A condition on the Extend relationship is optional. If no condition is indicated, the extension will always occur. It is not absolutely necessary for an actor to be linked to the extended use case.

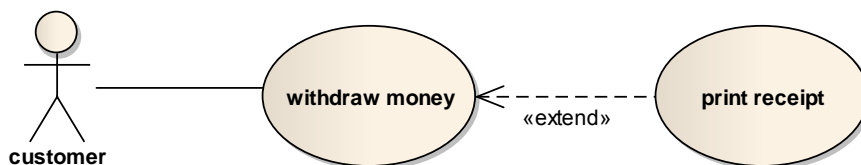


Fig. 11: Example «extend» relationship

The extended use case may be described more detailed by using extension points (Fig. 12). An extension point describes the event activating the extension. An use case may define several extension points. In addition to an extension point you may define conditions. If you don't specify a condition, the extension will be executed always. The example shows the use case *withdraw money* in rectangle notation³. The use case contains two extension points⁴. Both extension points are describing the sufficient trigger (logical “or”, one of them is enough). But there is also a condition *paper available*. At least one extension point for a must become true AND the condition *paper available* must be valid to execute *print receipt*!

As in the case of an *include* the diagram does not specify the timing circumstances. It may be found within the scenario(s) of the use case or in behavioral diagrams describing the use case in detail.

³ right click on element | Advanced | Use Rectangle Notation

⁴ right click on element | Advanced | Edit Extension Points...

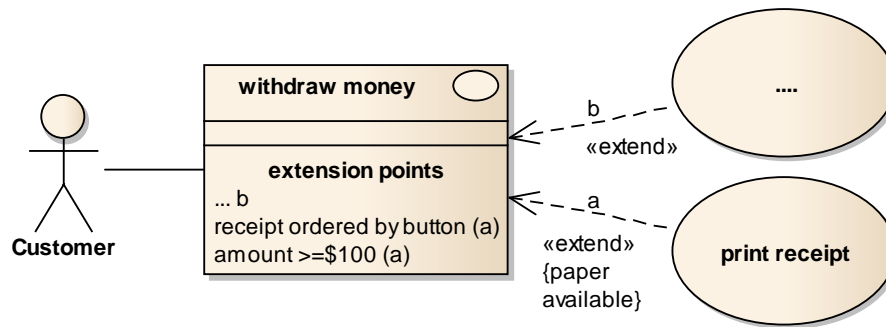


Fig. 12: Example «extend» with extension points and condition

If an use case is extended by several other use cases, you might reference the extension leg within the extension points by adding an index character.

Please pay attention to stay at the same abstraction level when defining extensions.

If a constraint has been defined, it has to be true to enable the extension.

Specialization (Generalization)

Another relationship is “Specialize”. A use case (or an actor) stems from a generalized use case (or actor) and specializes it. For example, saving for the actual distribution channel, a sale at the box office is similar to a sale via the Internet. It is possible to create a general use case, “sales”, and in specializing this use case to include the altered handling steps which occur due to the different distribution channels. This general use case is thereby assigned proxies for the various roles assumed by it. The same concept applies to actors.

Generalizations will be used for generic or abstract descriptions of functions too. The use case *perform Authentication* in the right sided figure is an abstract⁵ one and will not be performed itself. The two use cases *perform Authentication by finger print* and *perform Authentication by PIN* are two concrete variants of the generic Use Case. The *perform Authentication* may be used as a “placeholder” to stress that customers will have to identify themselves by choosing one of the variants. The abstract use case *perform Authentication* contains a generic description, what authentication has to provide, while the other use cases describe the deviations for the specific variant.

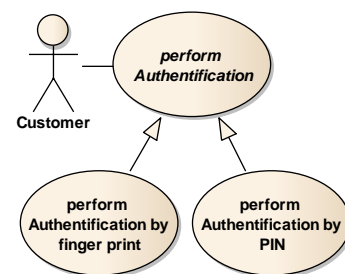


Fig. 13: Generalisation of Use Cases

An actor describes a role, which may be defined arbitrarily abstract. For example a customer of any bank may use *withdraw money*. If the bank operating the teller machine is the borrower’s bank, the customer may *deposit money* too. This may be modeled by an additional actor *Customer of TM-Operating Bank*. Due to the fact, that this customer is also a “standard” customer, he is allowed to use everything that *Customer* is allowed to use – *withdraw money*. In the diagram these circumstances are expressed by the generalization between *Customer of TM-operating Bank* and *Customer*. By this he becomes a *Customer* additionally (generalization is also named “is-a”), and he/she inherits *withdraw money* by this. On the other hand, *Customer* is not allowed to run the use case *deposit money*.

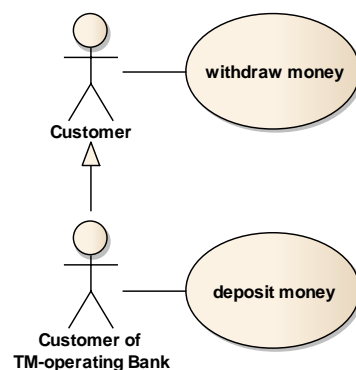


Fig. 14: Generalisation of Actors

⁵ select the element and within the window [View | Element Properties] set *Abstract* to *True* in the section *Advanced*

Hint: The opening triangle at the side of the generalist was chosen as symbol to indicate that the specialist has more functionality/capability, exceeding the functionality/capability of the generalist.

Descriptions and Notes

For all use cases and actors, UML allows descriptions to be added in the form of verbal and structured phrases. Due to their complexity, these are not suitable for display in diagrams. You can therefore add notes to the diagrams which refer to the key design concepts. Notes are displayed as a square, the upper-right corner of which is folded in. A dashed line establishes the link between the note and element to be explained.

To avoid parallel, conflicting comments – in diagram and within the element – you may put a reference to element content⁶ for the note in the diagram.

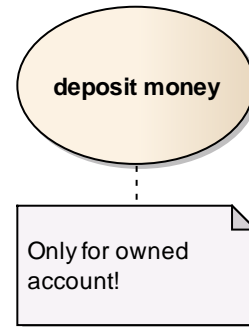

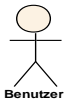

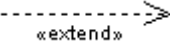
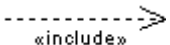
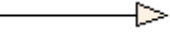
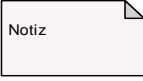
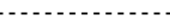


Fig. 15: Notes in Diagrams

Graphical Elements

The following table lists the symbols used in modelling a use case diagram:

Name/Symbol	Usage
Use Case 	A use case is illustrated as an ellipse containing the name of the use case. The name of the use case is typically formed by a noun and verb whereby the object to be manipulated and the activity to be carried out are described in a clear and concise manner. Using the <i>rectangle notation</i> ⁷ allows to display more details.
Actor 	A use case is triggered by an actor. Its illustration depicts a stick figure. Actors may also be placed within a square with the stereotype «Actor» placed above the name of that actor.
Use 	When an actor has triggered a use case, that actor has formed a relationship with that use case. This relationship is shown by a line connecting the use case and the actor.
Extended 	When a use case is extended by another under a specific condition, this relationship is indicated by connecting the use cases with an arrow labeled with the «extend» stereotype. The arrow points to the use case which is being extended.
Includes 	If one use case is contained within, and is therefore a key component of, a second, then both use cases are linked by an arrow labeled with the «include» stereotype. The arrow points to the contained use case.
Generalisation 	This relationship can be modeled between actors and use cases, and means that a use case or an actor is being specialized. The arrow points to the actor or the specialized use case.
Note  Note Connection 	Notes are diagram elements which are applied to other modelling elements. They contain information which provides a better understanding of the model, and are connected to the element by a dashed line.

⁶ select the element | Add | Note | OK (leave empty); right click on connector Link this Note to an Element Feature...

⁷ right click on element | Advanced | Use Rectangle Notation

Example

A customer wishes to withdraw money from an automatic teller with a bank card. The actor named *customer* plays this role and is the generalization for *own bank customer* and *third-party bank customer*. The specialized actors communicate via the role of the customer with the *identify card* use case which proceeds equally for both types of customer. This use case contains the use case *check account and PIN*, whereby the right of the customer to use the card is assessed. If an incorrect PIN has been repeatedly entered, the card is withdrawn. To model this, the use case *identify card* is extended by the use case *impound card*. This is executed only under the condition that the customer repeatedly entered the incorrect PIN.

Both actors, *own bank customer* and *third-party bank customer*, communicate directly (and not via the role of *customer*) with the use case *pay out*. This procedure varies between these two types of customer; i.e. the highest withdrawal amount and/or fees per transaction can vary.

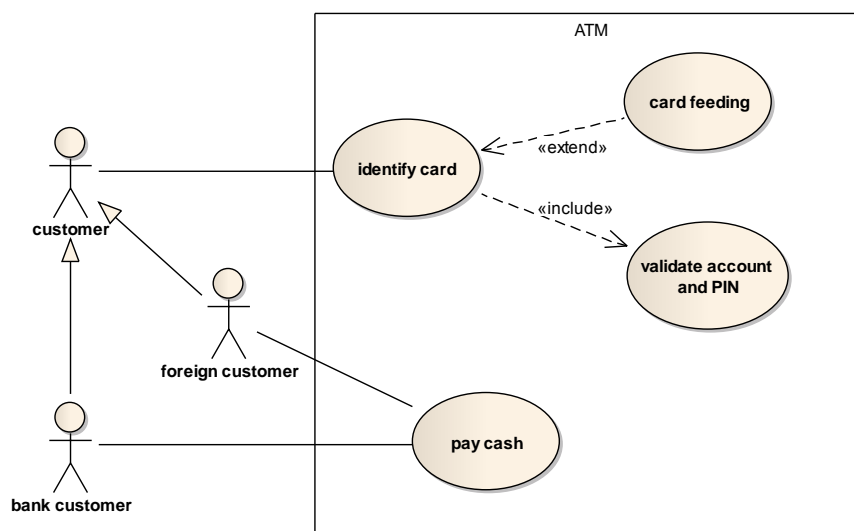


Fig. 16: Example of a Use Case Diagram

Chapter Review

Select the correct answers:

1. The Use Case Diagram

- [a] illustrates use cases of a software product in a chronological context
- [b] shows the structure of elements with methodologies and attributes
- [c] is used to define and communicate requirements on a software product

2. A person who operates a software product

- [a] slips into a role represented by an Actor
- [b] is indicated by name in the Use Case diagram
- [c] is the central figure in the Interaction diagram

3. A Use Case is represented by

- [a] a block symbol with names above it
- [b] an ellipse with names within and below
- [c] a rectangle with a colon before the name

4. An Include relationship states that

- [a] one use case must appear within another
- [b] one use case will probably appear within another
- [c] one use case represents a specialization of another

5. An Extend relationship states that

- [a] a Use Case may be processes together with another one optionally
- [b] a Use Case must be executed together with another one
- [c] a Use Case depends on another one an cannot be extended

Correct answers: 1c, 2a, 3b, 4a, 5a

Activity Diagram

Using Activity diagrams, chronological cycles can be graphically depicted as they are described in use cases. Single activities and their interdependencies are shown. Use cases can also be described with natural language, so-called scenarios, whereby the overview remains intact only for very simple processes. With Activity diagrams, however, it is possible to show even very complex processes with many exceptions, variations, branches and repetitions in a clear and coherent manner. In practice, it is now customary to cancel descriptions of scenarios expressly in diagrams in order to trace the contained expressions when covered during implementation, and to set up test cases.

The semantics of the individual model elements differs greatly from the model elements in UML 1.x despite the same terminology. In UML 1.x, the activity element yields to the action, whereas an entire activity model is now called Activity. A number of references to the UML versions can be found at the end of the chapter on Activity diagrams.

Activities

Activity describes the procedural order of actions. It is represented by a rectangle with rounded corners. The Activity's nodes and edges are located within the rectangle. In the upper left corner is the name of the Activity. Within an Activity you will find Actions. There are several types of Actions available: *normal* (an atomic working step), *CallBehaviourAction* and *CallOperationAction* for referencing behavior defined somewhere else. It's also correct to draw Activities within Activities – for better structuring.

As with every behavior in UML, an Activity can also have parameters. Inbound or outbound objects in an Activity model are identified as parameters of that Activity. These objects are placed on the Activity rectangle and also below the name of the Activity with type designations listed.

The following example shows an activity for the production of sixpacks. This activity has two parameters: an inbound parameter, *produced bottles* in the condition [empty], and an outbound parameter, *new sixpack*. The precise declaration of activity parameters are at the top left directly under the name of the activity.

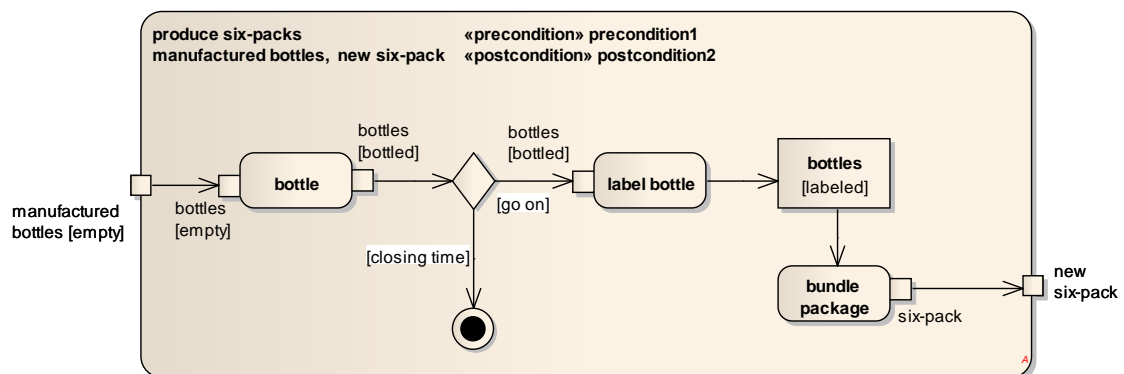


Fig. 17: Example of an Activity, "Production of Sixpacks"

This activity shows various kinds of nodes and edges. The rounded rectangles are actions. The small rectangles on the actions are so-called pins. They provide the entry and exit parameter values for the actions.

The Token-Concept for Activity-Diagrams

Up to UML 1.x Activity-Diagrams have been defined as a mixture of State-Diagrams, Petri nets and Process-Diagrams, leading to problems, practical and theoretical ones.

Starting with UML 2.x the *token semantics* of Petri nets has been applied, providing precise rules for the logical flow and flow of objects, including parallelization, synchronization and merging of

paths. A token corresponds to an executing thread, which can be generated and destroyed. The token represents the progress of the logical flow or of the data-/object flow. By this formal specification of the semantic of Activity-Diagrams it's possible to apply an automatic verification of Activity-Diagram, namely a simulation.

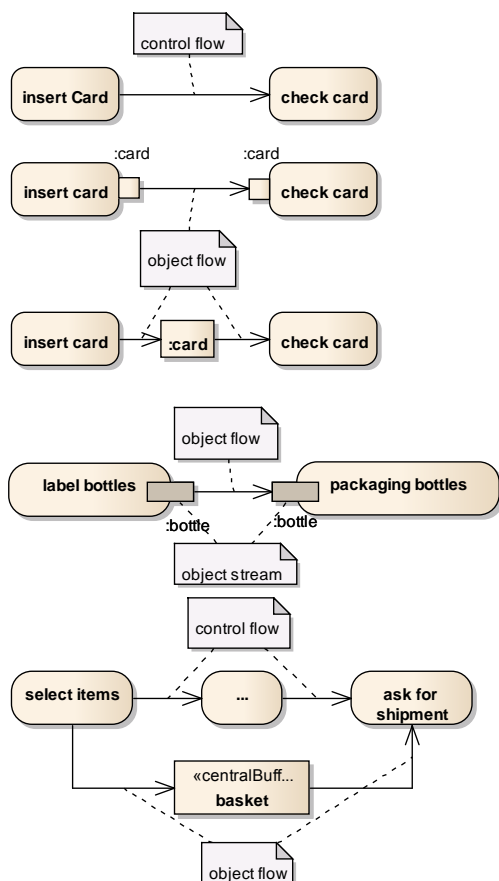
Due to the revision, a change of terms is in effect:

- The rudimental, indivisible steps are now named *Actions* (and not Activities).
- A collection of steps, means a complete Activity-Diagram or a part of it, are called now *Activities*.
- While in UML 1.x each token incoming by a transition started a working step, now an implicit synchronization is in progress, that means all incoming object- and control flows have to reach the element to start its behavior.
- Similarly the dropout of an Action or Activity occurs at the moment when all legs can be "fired". Formerly the legs have been named transitions and the author had to specify conditions, ensuring that only one transition was selected. Now the firing is delayed for all legs until all conditions for all legs become true.
- There are new elements:
 - Activities may have object nodes as input- or output parameters.
 - Pre- and post-conditions for Activities can be specified now.
 - Starting- and final activities are now named Initial node and final node.

Connections

The connections between actions differentiate between control flow and object flow. In notation both angles are the same: a solid line with open point.

Object flows can either be expressed with a straight line with an arrowhead between object-pins (a small rectangle on the boarder of an activity or action) or by pointing to or from an *object*, *datastore* or *central buffer node*.



A control flow connects actions and activities. When *insert card* is finished, the token moves through the control flow towards *check card* if *check card* is ready to be activated.

By an object flow, beside control data (or physical or logical objects) are transmitted. If several tokens are arriving, FIFO (first in, first out) is the applied rule. Instead of the pin-notion, an object symbol may be used.

An object stream is a special cases of object flow: a continuous flow of data (objects), like a conveyor band.

Object flows and control flows may be separated too. A *central buffer node* or a *data store* can be used for storing data temporarily or permanently.

Items may be stored in the basket (*Central Buffer Node*) and may be retrieved later. If the process is cancelled before the recall happens, data are lost – in contradiction to *datastore*.

Fig. 18: Control Flow / Object Flow

Junctions (Decision and Fork=Parallelization)

A program flow junction is created by using a diamond (decision) symbol. A decision may have any number of outgoing legs (usually at least 2). Showing up more legs, it's treated as a *switch*. Alternatively a switch can be expressed as a sequence of decisions - wasting editing time and space in the drawing, this is unusual. Each outgoing leg of a decision must have a *[Guard]* !

Hint: Guards must cover all possibilities and must not overlap.

Two outgoing legs with the guards $[x < 0]$ and $[x > 0]$... incomplete, $x = 0$ cannot be handled!

Two outgoing legs with the guards $[x \leq 0]$ and $[x \geq 0]$.. overlapping when $x = 0$, disjunct junction not possible → not correct!

Several outgoing legs on an activity or action without a guard define splitting (=parallelization)! To avoid a mix-up with the implicit junction, it's a usual style not to use the implicit junction at all but the decision (diamond) symbol. After a decision, always a disjunct selection is in progress.

Equivalent

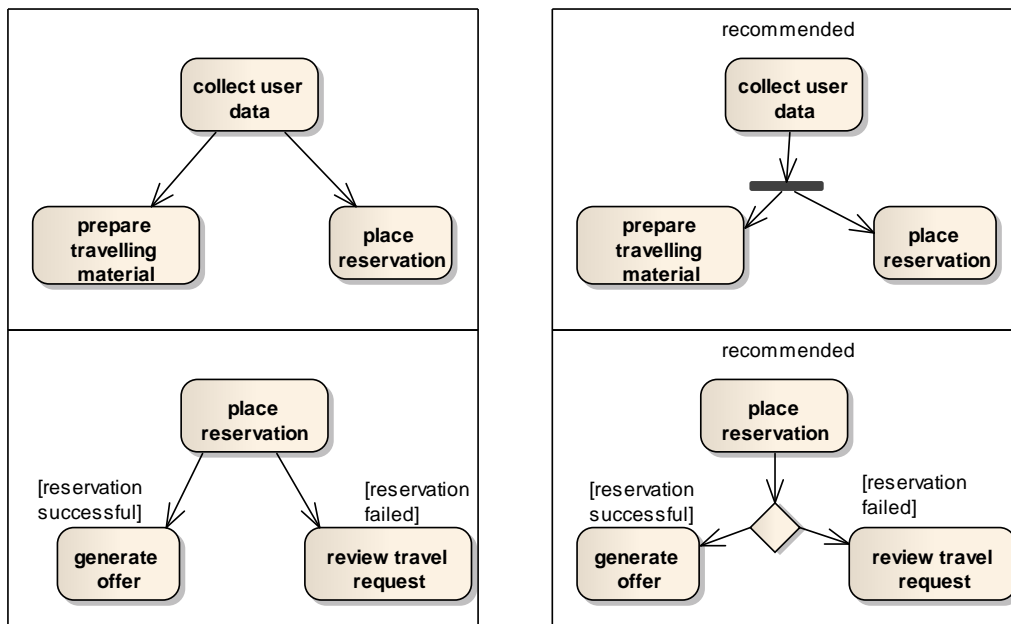


Fig. 19: Parallelization and Junction – implicit vs. explicit

Merge

By a decision an alternate path is selected. To realize a loop, a merge symbol is needed.

Connecting the *false* path directly to the activity *select appointed date* will simply be wrong: Two incoming lines are allowed in UML, but this expresses an implicit join – synchronization – on all incoming paths a token will be needed to start *select appointed date*!

Suggestion: Avoid implicit semantic at all.

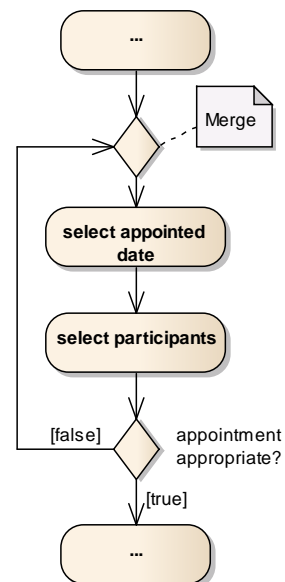


Fig. 20: Merging

Synchronization (Join)

Program flow is divided with a splitter (Fork, Parallelization symbol). The token hitting the fork generates a token for each outgoing leg, going their way now independently. Independence does not mean simultaneous – the behavior may happen simultaneous or not. If an outgoing token of the fork cannot be taken immediately by the subsequent element, its stored in a FIFO list until the subsequent action/activity can take it.

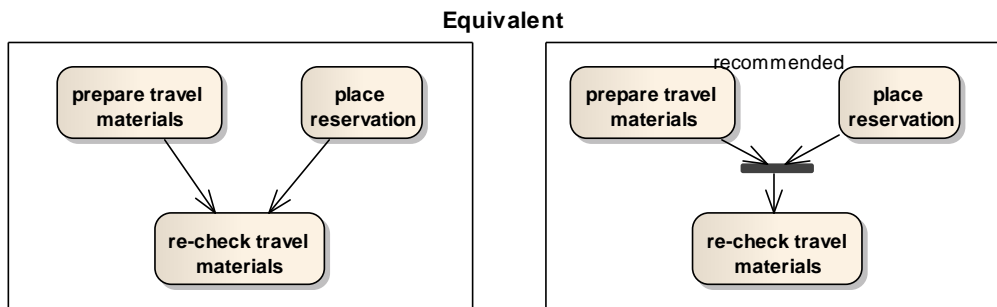


Fig. 21: Synchronization = Join

Independent processes can be consolidated by a *Join Node* (i.e. synchronization). A join node may have two or more incoming legs. For continuation it's necessary that on each incoming leg a controlflow- or objectflow-token reaches the join node. If this condition is true,

- all controlflow-tokens and identical objectflow-tokens are consolidated to one, singular token.
- All incoming objectflow-tokens are forwarded, but identical ones are consolidated into one each and are forwarded in a single instance only.

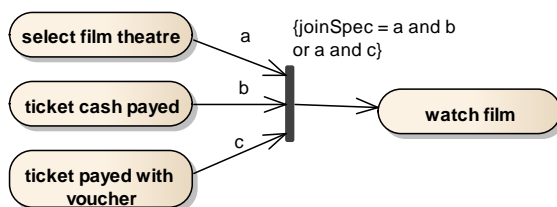


Fig. 22: JoinSpec

A join node has an implicit AND semantic. If only some of the arriving tokens shall be sufficient to continue with the synchronized path, UML provides the *Join Specification*⁸ (JoinSpec) feature. By this you may specify a condition, sufficient for synchronization. In our example, theatre selection and one of the payment methods are sufficient.

Composition of Activity Diagrams

Activities can be hierarchically composed. An action can be reconstituted from a number of detailed actions. The inbound and outbound edges of this constructed activity and the detail models must correspond. With this cascading of diagrams, one can retain an overview of more complex processes. This subcategorisation into sub- or detail-models can be helpful and also necessary in many regards: a) Adequate subdivision to maintain standard paper format, and b) Creation of detailed classifications that are included in various documents and approved by various responsible persons.

A call to an activity is represented by a *Call Behavior Action*.

Call Behavior Actions can be identified fork symbol in the lower right corner of the action.

In the figure *Enter PIN* beyond the Activity *update display* is called within the loop as well as after exiting the loop. *update display* is defined only once, but called by the actions several times.

To use this calling feature instead of copying the activity is strictly recommended. Please note: You can only call activities by *Call Behavior Actions*!

⁸ see Properties | Advanced

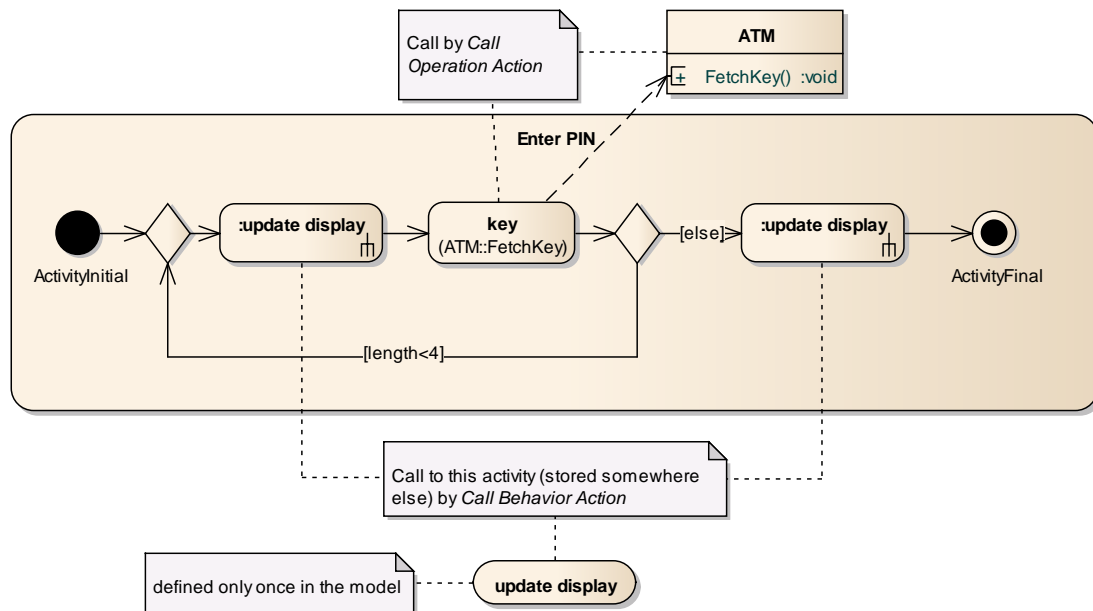


Fig. 23: Calling an Activity by an Action

Call Operation Actions are similar to *Call Behavior Actions*, not calling a behavior (activity) but an operation directly – the operation is defined somewhere else, for example as an operation of a class. In the figure above, the operation *FetchKey* of the class *ATM* is called by the action *key*. This is intended by several quality systems - like SPICE, CMMI, ... Independently this small effort will pay back later on, when a change request has to be implemented.

Enterprise Architect provides possibilities for structuring elements: A *structured (=composite) element* contains a link to a diagram where the reader will find detailed information concerning the element.

Graphically *composite* elements can be identified by a chain symbol in the right, lower corner.

This is not defined within the UML-specification, but offers a very powerful possibility to structure diagrams.

To cascade diagrams enables to achieve a good overview and an easy drill-down-feature to reach more details. This will be necessary and/or helpful to:

- a) deal with a limited printout format properly and
- b) to separate the content in a way, where different reviewers with different competency and/or authorization level can agree easily.

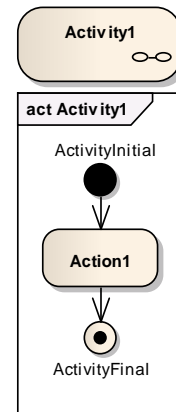


Fig. 24: Structured (composite) activities

Responsibility Zones (Swimlanes)

Single actions in the activity diagram are usually carried out by a responsible actor. In order to illustrate the allocation of the single actions to the actors in a diagram, so-called *Swimlanes* can be introduced. These vertical or horizontal lanes symbolize the actor and guide the activities through the graphical assignment of the individual actions to a lane in that actor's area of responsibility. Comment: This is represented only graphically, not logically!

Alternatively to *swimlanes* Enterprise Architect offers *partitions*, graphically very similar to *swimlanes*, but logically represented in the model. This allows a sorting of elements in the project browser by partitions, entering of detailed properties for the *partition* and – last but not least – pointing to an *instance classifier* carrying the definition of the task owner. The next figure gives an example of partition usage.

Asynchronous Processes

Controlflows or objectflows are connecting activities and actions. Processes defined by that are “synchronous” processes – determining the flow through possible subsequent working steps.

By usage of signals (*Send Signal Action*, *Receive Signal Action* and *Timer Action*) processes can be uncoupled, can be transformed into asynchronous processes.

By a *Send Signal* a broadcast signal is emitted. All *Receive Signals*, for which this signal is designated, will be activated. To enhance the readability *dependency connectors* may be used pointing from the *Receive Signal* to the *Send Signal* element!

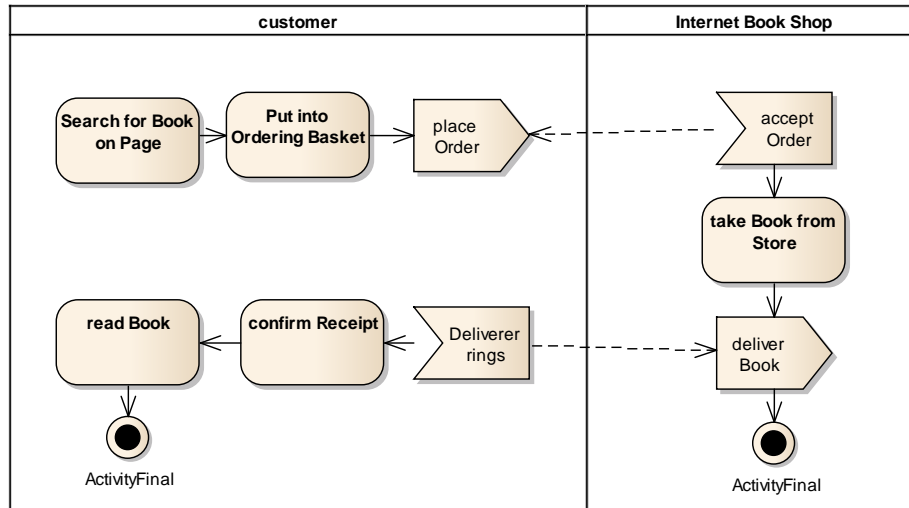


Fig. 25: Send / Receive

The figure above shows that *place Order* is a *Send Action* that is taken by the *Receive Action* named *accept Order* by the Internet Book Shop

Hint: *Receive Signal Actions* do not need to have an incoming leg. UML uses the rule, that any element without an incoming leg will earn a token when the activity diagram comes to life. The exception is the *interrupt region* – elements within such a region without an incoming leg will earn a token at the moment when the flow enters the region.

Interrupt Region

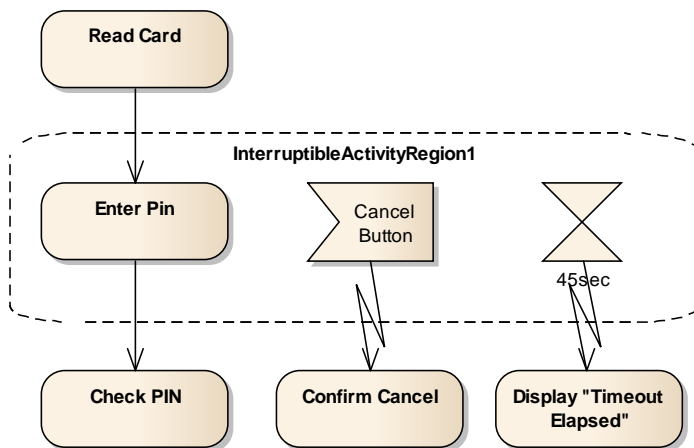


Fig. 26: Interrupt Region

An *Interruptible Activity Region* defines a special part of a process. When the region is entered by the token coming from *Read Card*, the process can be interrupted at any time. When the “normal” token crosses the border of the region behind the activity *Enter Pin* both receive symbols (*Time Event* is also a receive) lose their tokens.

If an interrupt reaches the Receive of *Cancel Button* or the time defined in the timer elapses, the “normal” token is destroyed and a token runs


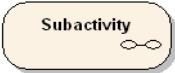
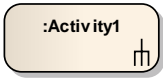
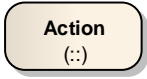
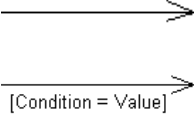

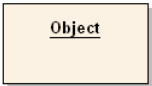
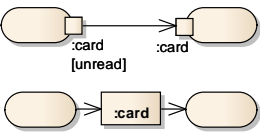
along the *interrupt flow* connector (lightning symbol) to the outside also killing


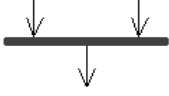



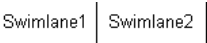



all tokens of the not yet fired receives within the region. Please note: The interrupt flow must lead



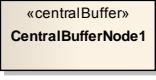
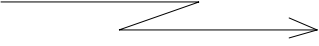
from an receive inside the region to an element outside of the region. UML provides no definition how the disintegration of the “normal” token is handled. Usually *disable/enable sections* are used in the detailed description of activities like *Enter Pin* to express if the interrupt is immediately in effect or will be cached until the enable-statement is reached.

In our example above *enter Pin* can be interrupted either by the cancel-button or by the elapse of the timer *45sec*.

Graphical Elements

Name/Symbol	Use
<p>Action</p> 	The Action symbol is a rectangle with rounded corners. By definition, an action is a single step which can't be further subdivided, nor be broken up by outside influences.
<p>Subactivities</p> 	The symbol for combined activities is the Activity symbol with two small activity symbols drawn into the lower right-hand corner.
<p>Call Behavior Action</p> 	A <i>Call Behavior Action</i> allows to call any behavior (activities) by this element. This will avoid redundant definitions of activities.
<p>Call Operation Action</p> 	A <i>Call Operation Action</i> calls a discrete behavior of a structure element, for example a call to an operation of a class or a use case or ... The names of the element and of the behavior will be shown as <Element name>::<behavior name> .
<p>Control Flow</p> 	Two actions are connected with an arrow when the activity flow changes from one action to another. The arrow points in the direction of program flow. The arrow can receive a condition as a label when the program flow takes place in this condition only. This is the case when numerous transitions emerge from one activity, or if flow is split by a diamond symbol.
<p>Junction, Merge</p> <p>Decision </p>	Program flow can either branch out or be re-united with a diamond symbol. Should one transition enter and several exit, then this is a Decision. Should several enter and only one exit, this is a Merge, in which case no labeling is used.
<p>Object Condition</p> 	Object Conditions are represented by a rectangle in which the name and condition of an object are given
<p>Object flow</p> 	An <i>object flow</i> describes the transmission of control from an action/activity to the next one and additionally transfers data/objects. This can be expressed by object nodes or an object between the actions/activities. Objects and object pins are instances of a class, a <i>Central Buffer Node</i> (transient buffer node) or a <i>Datastore</i> (persistent buffer node). Object nodes and objects may have a type (:card) and optional they may define an actual state ([unread]).

Name/Symbol	Use
<p style="text-align: center;">Splitter</p> 	<p>Using this element (Fork), program flow can be split into several program flows which run parallel. The entering token (control or object token) is duplicated for each outgoing path.</p>
<p style="text-align: center;">Synchronisation</p> 	<p>This element merges program flows which were separated by the splitter. A Synchronization can also take place which halts processing until all parts of the entire flow have arrived at the Synchronization element. This AND-semantic may be redefined by a <i>Join Specification</i>.</p>
<p style="text-align: center;">Start Point</p> 	<p>The entry point which starts processing after a use case is triggered is illustrated as a filled-in circle. When many start points are available, the concerned process branches are started parallel. If no start point is available, all nodes with no entry edge are interpreted as start points. To ensure proper understanding, one start point per process should be defined.</p>
<p style="text-align: center;">Activity Final</p> 	<p>After all actions of an activity have been processed, program flow ends this activity. This point is shown with an end point – its symbol is a small filled circle surrounded by a larger circle. An activity diagram may have as many end points (Activity Final) as desired; endlessly running processes must not have one; if one wishes to express the end of an activity at numerous points, then the paths must not be merged. Warning: The Token which ends up here is not cancelled, but rather returned to the superordinate element that was allowed to jump to the current diagram. The sub-process (<u>all</u> still running tokens in the diagram) will be terminated!</p>
<p style="text-align: center;">Flow Final</p> 	<p>Cancel, Flow Final, means that a Token reaching this symbol will be cancelled. The process branch is cancelled here. Should further Tokens exist, the entire process will be continued; if it's the final Token, then the entire process is ended.</p>
<p style="text-align: center;">Swimlanes</p> 	<p>To model program flow with actions not belonging to the same areas of responsibility, like to different packages, areas of responsibility (Swimlanes) can be modeled with vertical or horizontal lines. The name of this area between two lines is labeled at the top with the name of the responsible actors' instance.</p>
<p style="text-align: center;">Send Signal Action</p> 	<p>A <i>Send Signal</i> is an action used in a process to transmit asynchronous messages to other processes.</p>
<p style="text-align: center;">Receive Signal Action</p> 	<p>A <i>Receive Signal</i> is an Action waiting for a signal (event). At the moment the event is arriving, the defined action is performed and the flow is continued. <i>Receive Events</i> are used to model asynchronous behavior. If a <i>Receive Signal Action</i> has no incoming leg and the element carrying the <i>Receive Element</i> (region, diagram) is active, it's ready to "fire".</p>
<p style="text-align: center;">Time Event</p> 	<p>A <i>Time Event</i> generates an output (token) periodically. The output continues the subsequent flow. It may be used together with an <i>interruptable activity region</i>. If a <i>Time Event Action</i> has no incoming leg and the element carrying the <i>Time Event Element</i> (region, diagram) is active, it's ready to "fire".</p>

Name/Symbol	Use
<p data-bbox="236 241 488 309">Interruptable Activity Region</p> 	<p data-bbox="544 241 1362 365">An <i>Interruptable Activity Region</i> is an area which can be left by events (<i>Receive Events, Time Events</i>). Actually performed actions/activities will be stopped and the alternate leg (interrupt flow, lightning symbol) will be used.</p>
<p data-bbox="301 472 421 499">Datastore</p> 	<p data-bbox="544 472 1362 566">A <i>Datastore</i> is a persistent buffer node. It's used to take data/objects out of an object flow. By this you can express an access to already stored data or a persistent writing of data.</p>
<p data-bbox="245 651 480 678">Central Buffer Node</p> 	<p data-bbox="544 651 1362 801">A <i>Central Buffer Node</i> is a transient node. It has the same behavior as a <i>Datastore</i>, but the stored content will be destroyed when the activity ends – when an <i>Activity Final</i> is reached. This is the semantic of a local variable within an operation within OO-oriented programming languages.</p>
<p data-bbox="277 824 445 851">Interrupt Flow</p> 	<p data-bbox="544 824 1350 851">An <i>Interrupt Flow</i> is used to exit from an <i>Interruptable Activity Region</i>.</p>

Example

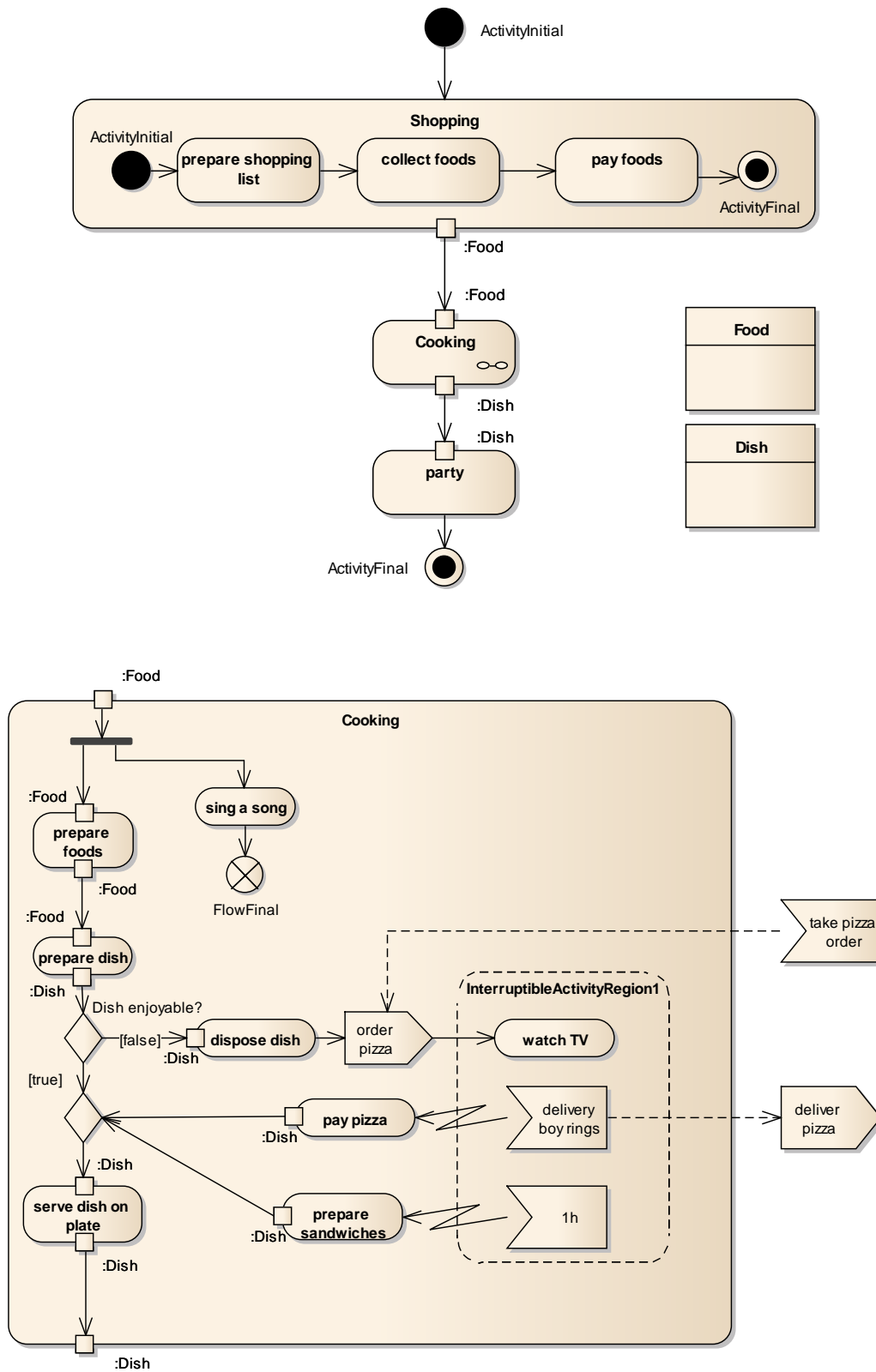


Fig. 27: Example of Activity: Prepare for Party

In the example above a process for preparing a party is modeled. The model has been divided into several structured diagrams.

The first diagram describes three activities processed sequentially: *Shopping*, *Cooking* and *party*. *Shopping* is modeled as structured activity, showing the subsequent activities inside of *Shopping*.

From *Shopping* an object flow transfers foods bought towards the activity *Cooking*. The types *Food* and *Dish* have been modeled as classes and may be described in detail there, for example be adding attributes like calories, weight, etc. .

Cooking has been modeled by a separate diagram. The activity *cooking* receives *Food* as Input and forwards *Dish* as an output. *Cooking* start with concurrent processes: *preparing foods* is performed while a song is sung. When the song ends, this thread is terminated.

The activity *prepare dish* transforms foods into a dish. If the dish is not enjoyable, it's disposed and a pizza will be ordered.

The ordering is a *Send Signal Action* covered by *take pizza order*, a *Receive Signal Action*. This triggers another process, not shown in the current diagram.

While waiting for the pizza, *watch TV* is started – within a *Interruptable Activity Region*, which can be left by the ringing of the delivery boy or by a timer set to 1 hour. Either the pizza is delivered within one hour – it will be paid and served then – or the timer elapses and some sandwiches are prepared and served, both by merging the flow at the merge node.

After putting the dish on the plate, the activity *cooking* is finished.

Hint: If the actual token position is outside of the interrupt region, all incoming signals will be ignored!

Chapter Review

Select the correct answers:

1. The Activity Diagram

- [a] shows the chronological order of actions and activities
- [b] has not changed very much since UML 1.4
- [c] shows the size of actions and activities relative to their duration

2. An Activity

- [a] can include additional activities which are hierarchically composed
- [b] is a process step which cannot be further divided
- [c] illustrates the communication between objects

3. Splitting and Synchronization are applied to

- [a] compare results of a decision
- [b] emphasize the time duration of actions
- [c] illustrate the parallelism of actions

4. Swimlanes are

- [a] areas of responsibility named after their responsible actor
- [b] areas of responsibility displayed within their actors
- [c] the conditional transition between two actions

Correct answers: 1a, 2a, 3c, 4a

State Machine Diagram

State Machine diagrams are not an invention of UML, but can rather be traced to David Harel's statecharts developed in the 1980's. This display format was taken on in UML.

A State Machine diagram shows a series of conditions which an object can take up over its lifespan, and the causes of the state changes. One can model the state and the changes in state of an object in dependency on executed operations. Special value is placed on the changeover from one state to the next. In this way, one can model an object from initialization to release. The State Machine diagram describes through which operations or events the conditions of the objects are changed. Furthermore, one can also see which configuration the attributes of an object have or must have before changeover.

An object can be modeled as a state machine diagram / -"system" as long as it can be given a list of states for which the following applies:

- The object is always (at every point in time of its existence) in a (1) state on this list; put differently:
- The object never finds itself in none of the named states (if so, then at least one state is missing on the list)
- Never in more than one state on the list (if so, then the state sub-categorization has been incorrectly chosen)

An object in a state can remain there, but it is also possible to specify "Activity" in states.

If an object is in a state, then sub-states can also be modeled for this state; for example, in a sub-ordered diagram (Composite Element/Child Diagram). If the behavior in a state of a procedural nature, then the sub-diagram can of course also be a state diagram of another kind.

State Machine diagrams must have a starting state and can have an end state. State Machine diagrams, so-called transitions, are always triggered by an event (e.g. requirement, timeout, etc.).

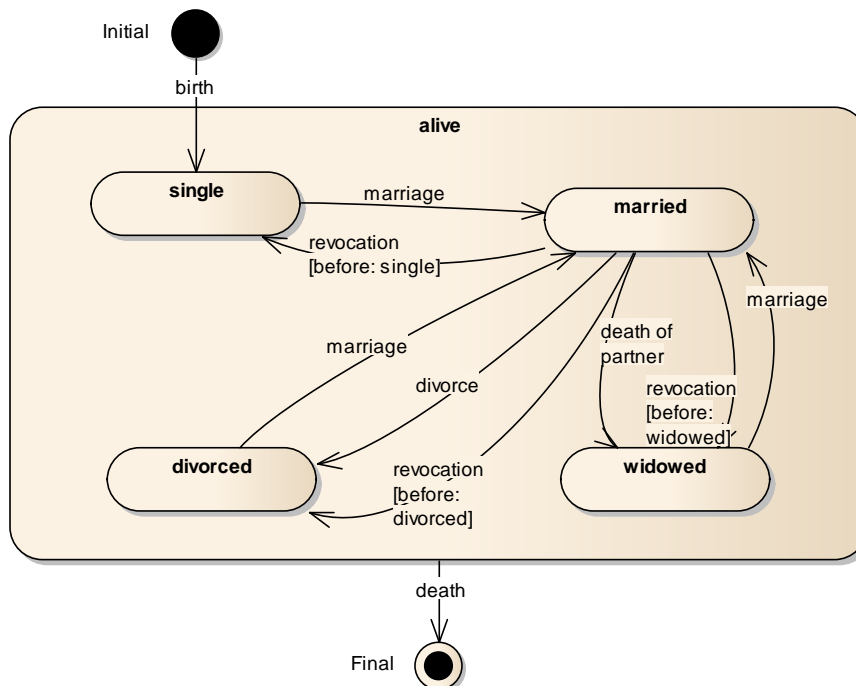


Fig. 28: Example State Machine Diagram

States

States are modeled using rounded rectangles. They can contain a name and, optionally, can be divided by horizontal lines in up to three areas. At the top is the name of the state. If the name is not entered then the state is anonymous. Existing state variables with value allocations typical for this state can be entered in another area. The third area within the state symbols can contain a list of internal events, conditions and resultant operations.

Event stands for three possible behavioral patterns:

- entry – triggers automatically when entering a state.
- exit – triggers automatically when exiting a state.
- do – is triggered over and over as long as the state isn't changed.

Transitions

Transitions from one state to the next are triggered by events. An event is made up of a name and a list of possible arguments. A state can place conditions on the event which must be fulfilled so that this state can be taken in by this event. These conditions can be independent of a special event.





An action can be carried out parallel to a state transition. The notation of a transition appears as follows:

Event [Guard] / Action

“[Guard]” and “/Action” are optional components - obviously. The listing of an event at the transition from the start point to the first state may be omitted. The event itself can also be left out on other transitions. If this is the case, then the state will automatically be changed when all activities of the previous state have been processed. The NO event (Trigger) is also designated as ANY Trigger - this event is ALWAYS present.

Symbols

The following table contains the state diagram symbols.

Name/Symbol	Usage
State 	The state of an object is symbolized by a rectangle with rounded corners. The State is named within this symbol.
Object Creation 	The start point of the state diagram is shown with a filled circle. It is identical with the object creation. Only one start point per State diagram is allowed and must be available. The location of the start point is optional.
Object Destruction 	The chain of state transitions ends with the object destruction. The end point is shown as a filled circle surrounded by a concentric circle. This symbol can be left out for endlessly running processes, but it may also be entered numerous times. Where applicable, the Token returns to the end of that activity in the super-ordinate diagram that called the sub-ordinate diagram.
Transition 	Transition is drawn by an arrow. The arrow is labeled with the name of the trigger that changes the object state. A Restriction [Guard] can be entered in brackets. This causes the object State to be changed only when this restriction has been fulfilled. Furthermore, behind a “/”, an activity list can be entered to be executed at transfer. Guard and activity lists are optional – even the trigger may be omitted on the transition from the Initial or if an ANY-Trigger is modeled.

Example

Startup of an automatic bank teller and main states. When switched on, the teller runs through a self-test. Depending upon the result, either the normal state or the error state is engaged. It has also been determined that, in case the self-test require too much time, that also here the error state is engaged. When a card is inserted, it is examined. Depending on the result, the machine continues to either the PIN-query or the cancel state. Further states such as account balance query, availability of funds, etc. are not shown here.

The chain symbols show that there are sub-diagrams that more precisely describe the behaviors in the states. Sub-diagrams can freely use as many Behavior diagrams as desired - these must not necessarily be additional State Machine diagrams.

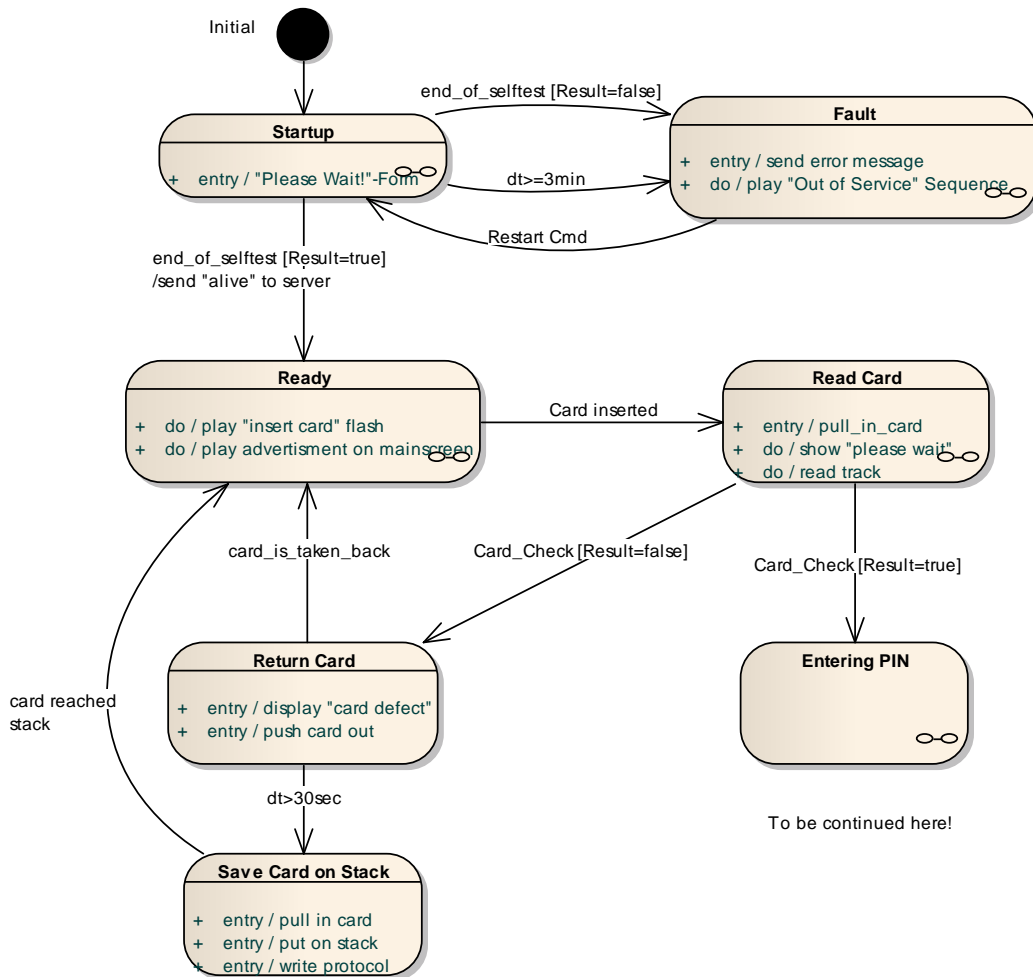


Fig. 29: Example State Machine Diagram “Automatic Teller Start-up”

Chapter Review

Please identify the correct answers:

1. A State without a noted name is
 - [a] Non-UML compliant
 - [b] an incomplete state
 - [c] an “anonymous” state

2. The start point of the State Diagram
 - [a] Is the first event which occurs
 - [b] defines a guard for the first state change
 - [c] is equivalent to object creation

Correct Answers: 1c, 2c

Class Diagram

The class diagram is the heart of UML. It is based on the principles of object orientation (abstraction, encapsulation, heredity, etc.) and due to its versatility can be implemented in all phases of a project. In the analysis phase it appears as the domain model and attempts to provide an image of reality. The software is modeled with it in the design phase (data structures), and in the implementation phase source code is generated (code structures).

Classes and the relationships of classes to each other are modeled in class diagrams. The relationships can be roughly divided into three categories. The simplest and most general option is the association. A second relationship which can be modeled is the acceptance of one class into another class – the so-called Container class. Such relationships are called Aggregation or Composition. A third option is Specialization or Generalization.

Since a class must model the structure and the behavior of objects which are created from this class, it can be equipped with methods and attributes. Furthermore, the modeling of base classes and interfaces can be achieved via stereotypes. Template classes can be implemented in several programming languages. UML displays such classes as parametrisable classes in the class diagrams.

Class

A class describes a number of instances which have the same attributes, constraints and semantics.

Classes are represented by rectangles which either carry only the name of that class, or also the attribute and operations. The three compartments – Class name, Attributes, Operations – are each divided by a horizontal line. Class names usually start with a capital letter and are mostly substantive in singular (collection classes, among others, in plural where applicable).

The attributes of a class are noted with at least their names, and can contain additional data pertaining to their type, an initial value, attribute values and constraints. Methods are also noted with at least their name, as well as with possible parameters, their type and initial values, as well as possible attribute values and constraints.

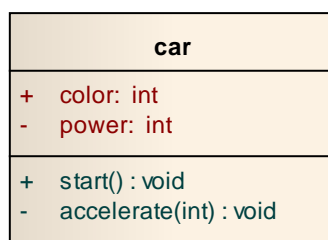


Fig. 30: Class Example

Scope

The Scope of class elements is labeled with a sign in front of the name. If an element is clearly visible, a "+" sign is shown in front of the name. Private elements are given the "-" sign. A "#" in front of a name means that the class element is labeled with the access attribute "protected". Protected is an extension of private; sub-(daughter) classes inherit attributes marked as protected. A "~" in front of the name means "package", a limitation of "public" - not unlimited public visibility, but rather limited to the package.

Abstract Class

Instances are never created from an abstract class. This class is intentionally incomplete and constitutes the basis for further subclasses which can have instances. An abstract class is illustrated like a normal class, however the class name is set as cursive.

Stereotypes

Abstract classes, for example, can be indicated with stereotypes. The specification of the stereotype appears over the class name in French quotations: « ». Stereotypes can also be made visible with various colors or by writing the class name in italics.

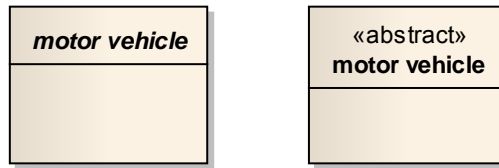


Fig. 31: Example Stereotypes

Parameterized Classes

A special kind of class is the parameterized class. Here, the type of contained attribute has not yet been established. Definition takes place when an object in this class is instantiated. The graphical appearance is modified for such classes. The classes rectangle is given a second rectangle with border at the top in which the variable type is shown.

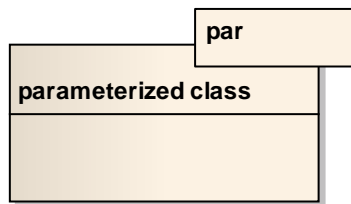


Fig. 32: Parameterized Class

Object

Objects are the operative units of an object-oriented application. They are created in memory according to a building plan – the class definition. Every object has its own identity. An object possesses a specific behavior which is defined by its methods. Two objects of the same class have the same behavior. Furthermore, objects have attributes which are the same as other objects in the same class. The state of an object is defined by its values which are saved in the attributes. Therefore, two objects in a class are equal when the values in their attributes correspond.

Like classes, objects are drawn in the diagram using a rectangle, although the name is emphasized to differentiate it from classes. The name of the object follows the class name separated by a colon. If the actual object name has no bearing for the case to be modeled, then it can also be omitted, whereby only a colon and the class name are shown. As the methods are not important for the object's presentation, they are not shown.

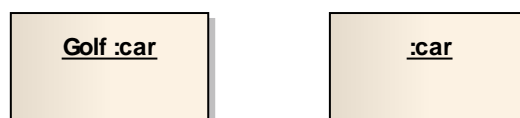


Fig. 33: Object Example

Attributes

An Attribute is a data element which is similarly contained in every object of a class, and is represented in every object by an individual value.

In contrast to UML 1.x, UML 2.0 no longer strictly differentiates between Attribute and Association ends. This means that presentation as Attribute in a class or as navigable Association is the same.

Every attribute is indicated by at least its name. In addition, type, visibility and an initial value can be defined. The full syntax is as follows:

[Visibility][[/]Name[:Type][Multiplicity][=InitialValue]

Methods (Operations)

A class must have a responsible Method for every message which it receives. A class provides other classes with functionality via a method. Using messages, or Method Calls, the objects instanced from the classes communicate with each other and achieve thereby coordination of their behavior. Objects and their communication via Method Calls are illustrated in the group of interaction diagrams.

Relationships

There are four different kinds of Relationships between classes, whereby generalization is a special form which is very similar to the other three – association, aggregation and composition.

Association

An Association represents the communication between two classes in a diagram. The classes are connected with a simple line. With the help of an arrow, a directional Association is shown.

Every association can be furnished with a name which provides a more detailed description. The name can also be furnished with a reading direction indicator – a small filled triangle. This indicator refers strictly to the name and has no relation to the navigability of the association.

On every page of an association, role names can be used to more precisely describe which role the current objects play in the relationship. Roles are the names of attributes which belong to the association or one of the involved classes.

In programming language, associations are generally realized in that the concerned classes contain relevant attributes.

A role therefore represents an attribute. Aside from role names, visibility specifications can be placed on every side of the association. If, for example, an association end is listed as private (-), then the object itself, or the object's operations, can utilize the association, whereby neighboring classes receive no access.

A Directional Association is noted like a typical association, except that on the side of the class to which navigation is possible – in the navigation direction – is an arrow with open point. Multiplicity and role names can theoretically also be noted on the side to which navigation is not possible. They describe a *Property* which does not belong to a class, but rather an association.

In the following example, the class Customer would receive an attribute "account" as reference to an object of the class CustomerAccount, and the class CustomerAccount would receive the private attribute "bpos" with a collection or subclass thereof which references the booking position objects.

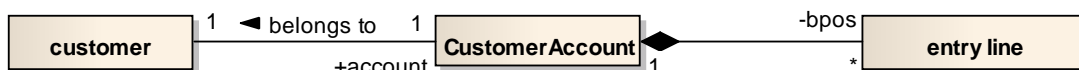


Fig. 34: Association and Composition with all properties

Many modeling tools use the role names of the relationship for the corresponding automatically-generated attributes; role names in UML also correspond formally with the corresponding attributes. Association with navigability is an alternative notation for attribute presentation in the appropriate class.

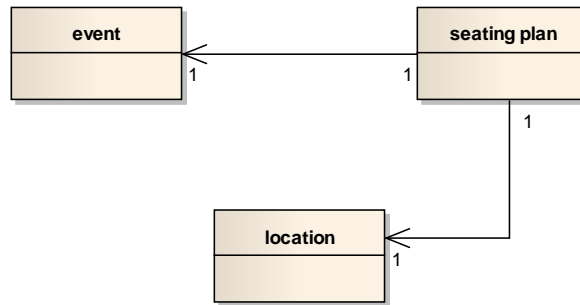


Fig. 35: Associations

These relationships are read in the following way:

- An event has a seating plan.
- A seating plan is allocated to a venue.

The arrow shows that communication emerges predominantly from the seating plan (the class therefore receives a reference to the venue at implementation). The cardinalities are thereby read before the target classes.

Multiplicity

The Multiplicity of an association indicates with how many objects the opposing class of an object can be instantiated. When this number is variable, then the bandwidth – or minimum and maximum – is indicated 3..7 . If the minimum is 0, this means that the relationship is optional (the relationship is still there, only the number of elements is 0). If the minimum is equal to the maximum, only one value is written.

In UML the term Cardinality is used to express the concrete number of instances.

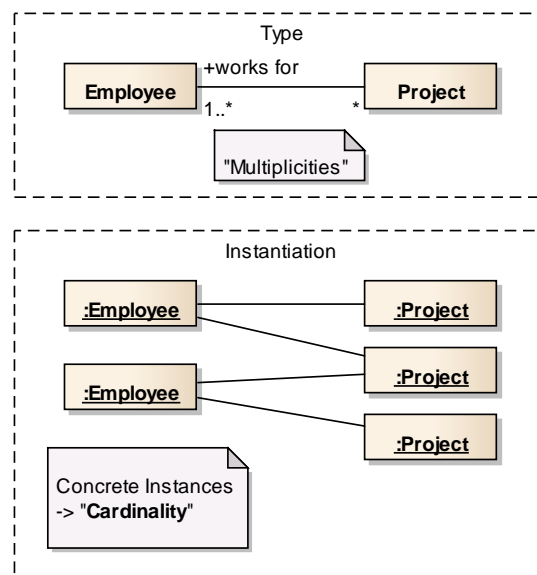


Fig. 36: Multiplicity vs. Cardinality

The term *cardinality* has its origin in data modeling – there with the meaning of multiplicity of UML. Within UML both terms are used to distinguish between the possible number of instantiations and the concrete number of instantiations. The cardinality describes within an object diagram the number of associated objects.

Association Class

Classes can also be arranged with each other in combination; when regarding the relationship of customer, account and bank card, then it becomes clear that a bank card exists for a combination of a customer and an account - and that is very different than associating the card directly with the customer and the account. UML provides Association Class notation for this:

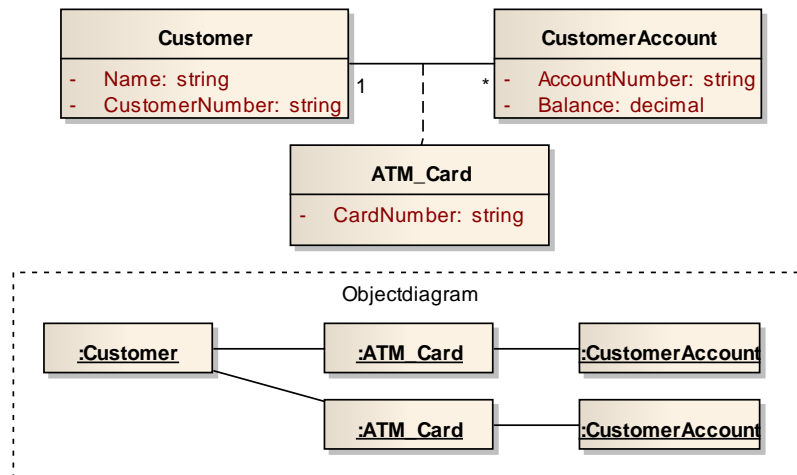


Fig. 37: Association Class

A unique property of the association class is that it may only give precisely one instance of the associated class per reference. For example, there may only be one reference of bank card per combination of customer and account.

Should several classes be involved, then the association node (n-ary Association) is used:

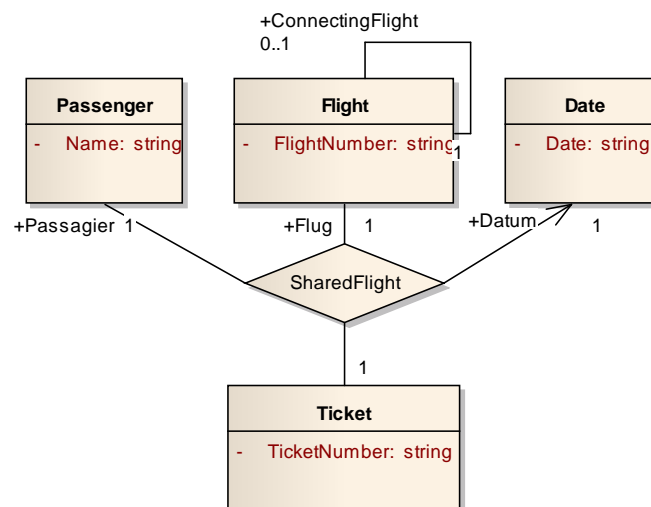


Fig. 38: Association Node

The n-ary association has its beginnings in data modeling (Entity Relationship Models) and is implemented in UML semantically identical. All classes involved with the association form a single unit. This means, for example, that (ticket, date, flight, and passenger) all form a single component and as a sum have a significant meaning: a role in a specific flight. This significance is usually expressed in the name of the association (hash).

Important for n-ary association is the setting of multiplicities, or what combinations of object characteristics are valid. In order to correctly set multiplicities, think for n-1 elements a multiplicity of 1 and set the multiplicity of the n-th.

For example: A passenger has precisely one ticket for a flight on a specific date. If it is possible that the passenger may have more than one ticket for the same flight on the same date, then the multiplicity for Ticket must be greater than 1 (e.g. “*”).

Aggregation

An aggregation is an association extended by the semantically non-binding commentary that the involved classes are not in an equal relationship, but rather represent a "part of" relationship. An aggregation should describe how something whole is logically made up of its parts.

Like an association, an aggregation is shown as a line between two classes and is given a small diamond. This diamond is on the side of the aggregation, or whole. It effectively symbolizes the container object in which the individual parts are collected. All association notation conventions are valid.

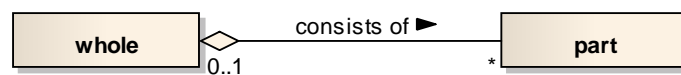


Fig. 39: Aggregation notation

The Company-Department-Employees example shows that a part (department) can at the same time also be an aggregation.



Fig. 40: Example Aggregation

Composition

A Composition is a strict form of aggregation whereby the existence of the parts depends on the whole. They describe how something whole is made up of individual parts. Like aggregation, composition is shown as a line between two classes and is given a small diamond on the side of the whole. Unlike that of aggregation, this diamond is filled in.

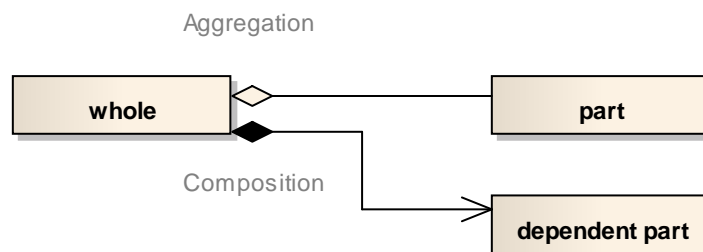


Fig. 41: Aggregation and Composition

When a class is defined as part of several compositions, such as in the following example, then this means that both cars and boats have a motor – a specific motor object but which can always only either belong to a car or to a boat. The decisive characteristic of a composition is that the aggregated parts will never be shared with other objects. This can also be written into the diagram using XOR constraint.

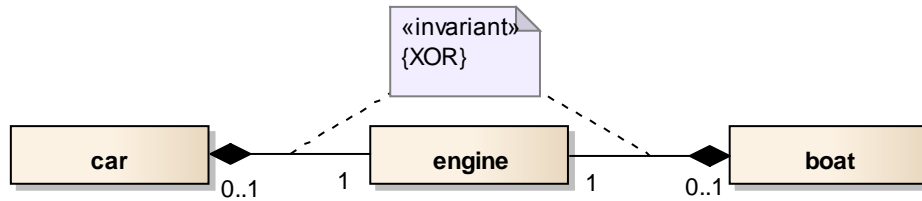


Fig. 42: Example Composition

Should the multiplicity for parts (Motor) start from zero (0..*), then the whole may exist without parts, or can contain as many parts as desired. Should the multiplicity of the whole (car/boat) be from zero to one (0..1), then that part may also exist without a whole; as soon as that part belongs to a whole it will no longer be separated and an existence-dependent relationship exists. Existence dependency means that the part without its whole cannot exist, and when the whole must be destroyed, then so must all of its parts.

In the following example, an aggregation is modeled between the class Event Plan and the class Seating Plan. A composition is defined between the classes Seating Plan and Seat. There is therefore no Seating Plan object without a Seat. The relationships are read in the following manner:

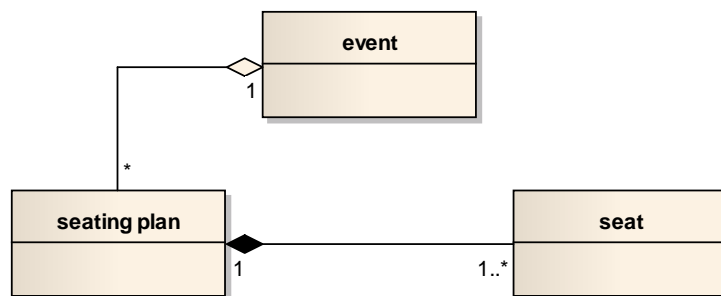


Fig. 43: Example Aggregation and Composition

- A seating plan has unlimited capacity, but usually only one seat.
- A seat belongs to one specific seating plan.
- Every seating plan belongs to one specific event.
- An event may contain an unlimited number of seating plans.
- The seating plan can also be assigned to another event (Aggregation), but must in any case always have one event.
- The seat is part of a seating plan; this relationship can never be changed (Composition).

Generalization/Specialization

A Generalization is a relation between a more general (parent) and a more specific (child) element, in which the specific element provides additional properties but is still compatible with its general element.

In generalization or specialization, characteristics are hierarchically arranged; this means that characteristics of general importance are assigned to elements of a more general nature, and more specialized characteristics are assigned to elements which are subordinate to more general elements. The characteristics of the parent elements are passed on ('handed down') to the appropriate sub-elements. A sub-element thus carries its specified characteristics as well as the characteristics of its parent elements. Sub elements therefore inherit all characteristics of their parent elements and can extend these further or overwrite them.

Hierarchical inheritance trees are modeled in this way. Inheritance trees are an important design element in the modeling of software architectures. The following example models how events can be modeled as in-house or external.

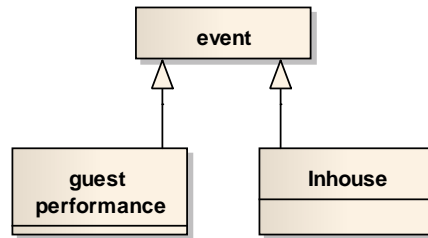


Fig. 44: Example Inheritance

Dependencies

A dependency is a relationship from one relationship of one (or more) source element(s) to one (or more) target element(s).

The target elements are required for the specification or implementation of the source elements. A dependency is shown as a dashed-line arrow pointing from the dependent to the independent element. Additionally, the type of dependency can be more precisely specified using a key word or stereotype.

Since, in a dependency relationship, the source element requires the target element for its specification or implementation, without the element it's incomplete.

Dependencies can have differing causes. Some examples are:

- One package is dependent upon another. Here, for example, the cause can be due to the fact that one class in the one package is dependent upon another class in the other package.
- A class is using a specific interface of another class. When the provided interface is changed, the class using this interface also requires changes.
- One operation is dependent upon another class; for example, the class is used in an operation parameter. A change in the class of the parameter possibly requires a change in the operation itself.

Stereotype	Meaning
«call»	Stereotype for Usage Dependency (Usage) The Call relationship is defined and specified from one operation to another operation so that the source operation calls up the target operation. A class may also be chosen as a source element. This means, then, that the class contains an operation which calls up the target operation.
«create»	Stereotype for usage dependency (Usage) This dependent element creates copies of the independent element. The Create relationship is defined between classes.
«derive»	Stereotype to define abstraction relations (Abstraction) This dependent element is derived from the independent element.
«instantiate»	Stereotype to define type instance relations (Usage) This independent element created copies of the independent element. The relationship is defined between classes.
«permit»	Keyword for the Permission relationship (Permission) This independent element has the permission to use private attributes of the independent element.
«realize»	Keyword for the Realization relationship (Realization) This independent element implements the independent element, for example an interface or an abstract element; or an element must implement a requirement.
«refine»	Stereotype to define semantic specialization (Abstraction) This dependent element is on a more precise semantic level than the independent element.

«trace»	Stereotype to define traces between semantic specializations (Abstraction) This dependent element leads to an independent element to be able to follow semantic dependencies, for example from a class to a requirement or from a use case to a class. Also used between a test case and the tested element.
«use»	Key word for Usage relationship This dependent element uses the independent element for its implementation.

The Abstraction relationship is a special dependency relationship between model elements on various abstraction levels. The relationship is noted as a dependency relationship. They cannot be confused, however, as the abstraction is always used together with a stereotype. UML defines the standard stereotypes «derive», «trace» and «refine».

A mapping also always belongs to an abstraction relationship showing how the involved elements interact (Mapping). Specification can be formal or informal. In spite of arrow direction, an abstraction relationship can also be bidirectional according to stereotype and mapping, e.g. the often-used «trace» relationship.

A special abstraction relationship is the *Realization* relationship. It is a relationship between an implementation and its specification element.

The *Substitution* relationship is a special realization relationship. It describes that examples of the independent elements can be substituted at run time from the dependent elements, for instance due to similar interface implementation.

The *Usage* relationship is a special dependency relationship. The difference to the dependency relationship is that the dependency limits itself only to the implementation and is not valid for the specification. This means that the dependent element requires the independent element for its implementation. Therefore, there can be no usage relationships between interfaces, but rather the dependency relationship.

The permission relationship is a special dependency relationship which issues the dependent element access rights to the independent element.

Interfaces

Interfaces are special classes which specify a selected part of the externally visible behavior of model elements (mainly of classes and components) with a set of characteristics (features).

The Implementation relationship is a special realization relationship between a class and an interface. This class implements the characteristics specified in the interface.

Interfaces are noted like classes; however, they incorporate the key word «interface».

A differentiation is made between available and required interfaces. An available interface is offered by a model element and can be used by others. A required interface is an interface which is requested by another model element.

Interfaces specify not only operations, but also attributes. Interfaces may also have appropriate associations to other interfaces or classes.

Classes which want to implement an interface must implement all operations defined in the associated interface. Concerning the attributes defined in the interface, the implementing class must behave in such a way as though it owns the attributes. In the simplest case, it actually implements the attributes. It is also enough, for example, to offer only the appropriate get() and set() methods.

A class can implement many interfaces and can furthermore contain additional properties, or put another way: As a rule, an interface describes a subset of the operations and attributes of a class.

A special realization relationship exists between the implementing class and the interface – the implementation relationship (key word «realize»).

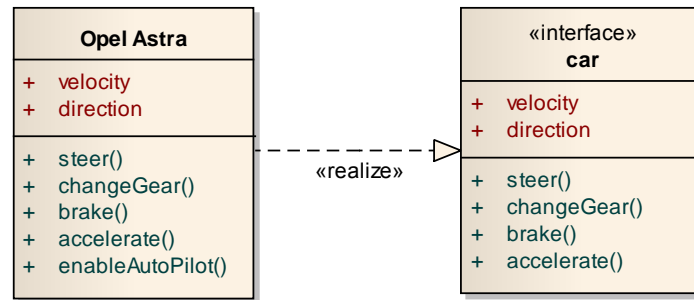


Fig. 45: Example interface

Given the current specification, it is not clear if, for the implementation relationship, the dashed arrow with keyword `«realize»` is to be used, or the dashed Generalization relationship as in UML 1.x. We assume here the latter, as this alternative is also much better for practical work.

The implementation relationship is noted with a special arrow. The other way to show the implementation of an interface is a small, unfilled circle connected by a line with the class that provides the interface. This should symbolize a plug. Next to this, the name of the interface is given; it corresponds to the name of the associated interface.

When drawing the arrow, the possibility exists to read the operations and attributes requested via the interface. The short notation does not indicate the operations and attributes requested by the interface, only the name of the interface. You may still use the 1.4 Notation, shown on the right side alternatively. The benefit: Available methods can be shown directly in the diagram.

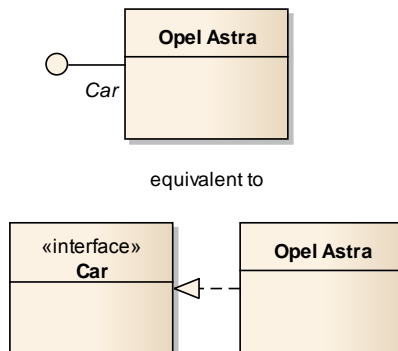


Fig. 46: Notation for Available/Provided Interface

For a requested interface, the requesting element has a usage relationship to the interface. The existing short notation is an open semicircle on a stem which is intended to symbolize a grabbing hand.

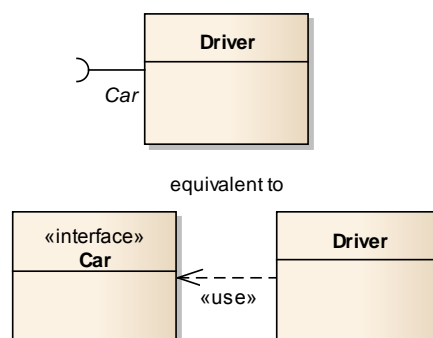


Fig. 47: Notation for Requested/Used Interface

Allocation and request of an interface can be shown by the combination of both short notations by sticking the plug into the socket.

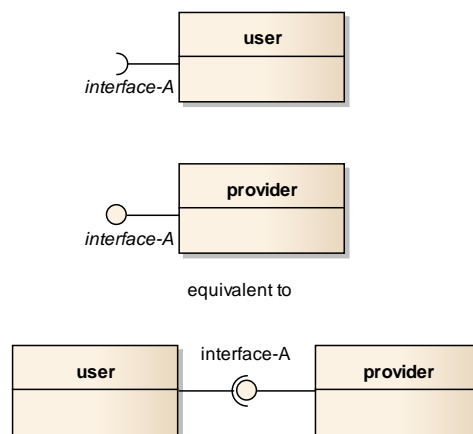


Fig. 48: Notation Options for Interfaces (Usage and Realize)

Warning: The last drawing version uses the symbol *Assembly*. But an assembly belongs to the category *connector*! You cannot use linking; the name can only be inserted textually! A connector cannot be an instance of an element, so there will be no feature to set the *instance classifier*! Avoid this kind of notation!

Since interfaces are special classes, all corresponding rules apply. Generalization is one option, for example.

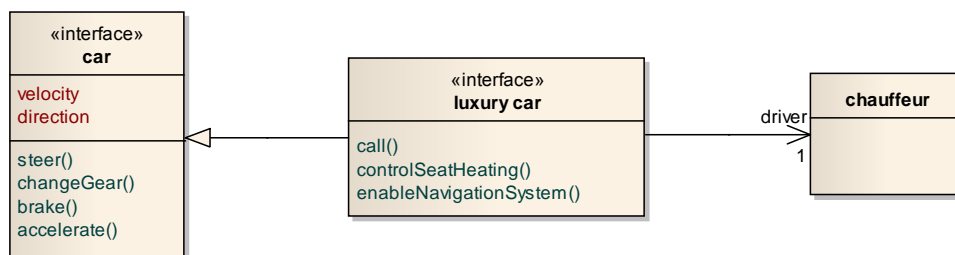


Fig. 49: Extension of Interfaces

The following image shows that the class “Opel Astra” implements the “car” interface. The car interface requests four operations – steer(), shift(), brake() and accelerate() – as well as two attributes, “velocity” and “direction”. The class “Opel Astra” fulfills the requirements for this interface, since it has, among other things, the four requested operations.

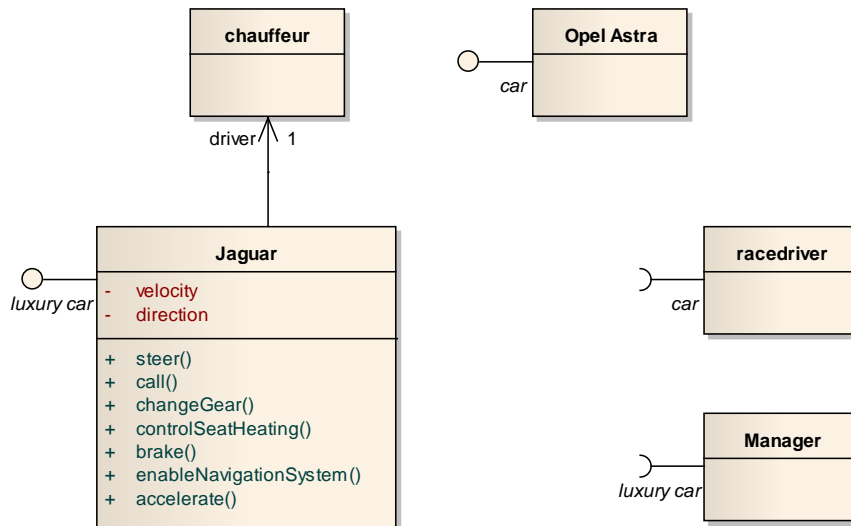
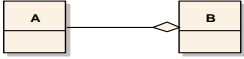
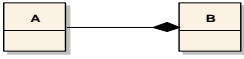

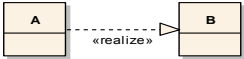


Fig. 50: Example of Implementation of Interfaces

Symbols

The following tables contain the symbols of the class diagrams.

Name/Symbol	Use
<p>Class</p>	<p>A class is indicated on a diagram with a rectangle in which the name of the class is shown. The class symbol can be extended by an area to indicate the attributes and the methods. For parametrisable classes, the class symbol is extended by a rectangle which contains the name of the parameter.</p>
<p>Object</p>	<p>Objects are indicated by a rectangle in which the name and the class of the object are underlined. Both are separated by a colon. The class name can be left off if the object can also be classified without a class name. An anonymous object (without name) is shown with only a colon and the class name.</p>
<p>Package</p>	<p>Packages include multiple classes which are grouped there for a specific task. The graphic for a package symbolizes an index file card on which the package name is shown.</p>
<p>Assoziation</p>	<p>If two classes are in a relationship, this is shown by the connection of both classes by a solid line.</p> <p>The cardinality can be entered on the line near the class symbol.</p> <p>Furthermore, the names of the relationship and role can still be shown (Roles: Accused, Accuser; Relationship Name: Accuses).</p> <p>You can indicate the preferred direction of communication using an arrow. (Navigation)</p>

Name/Symbol	Use
Aggregation 	Aggregation is shown with a line, on the end of which is a diamond pointing to the container class (Class B can manage objects of Class A). The diamond is not filled. Roles, cardinality and desired direction of communication are entered at the association.
Composition 	Unlike aggregation, composition is shown with a filled diamond. Further symbols correspond to aggregation.
Generalisation 	The symbol is an arrow pointing to the base class and connecting it to the derived class. The arrow is not filled (class A inherits from class B).
Realisation 	When a class imports an interface, the link is shown by an arrow on a dashed line.

Example

All performances in a theatre are managed using an event plan in a ticket system. The class event calendar is connected with the class seating plan via a composition. In this way, the event calendar class can manage objects of the seating plan class. Every seating plan contains the theatre's seats, and these are sold via the seating plan. The class seating plan manages the class seat via an additional composition. The navigation points in the direction of the class seat. The class seating plan communicates with the class event, where the data from the performances are stored, e.g. date and time. Entries for the venue, e.g. the name and the address, are stored in the class location. Navigation is carried out from the class seating plan.

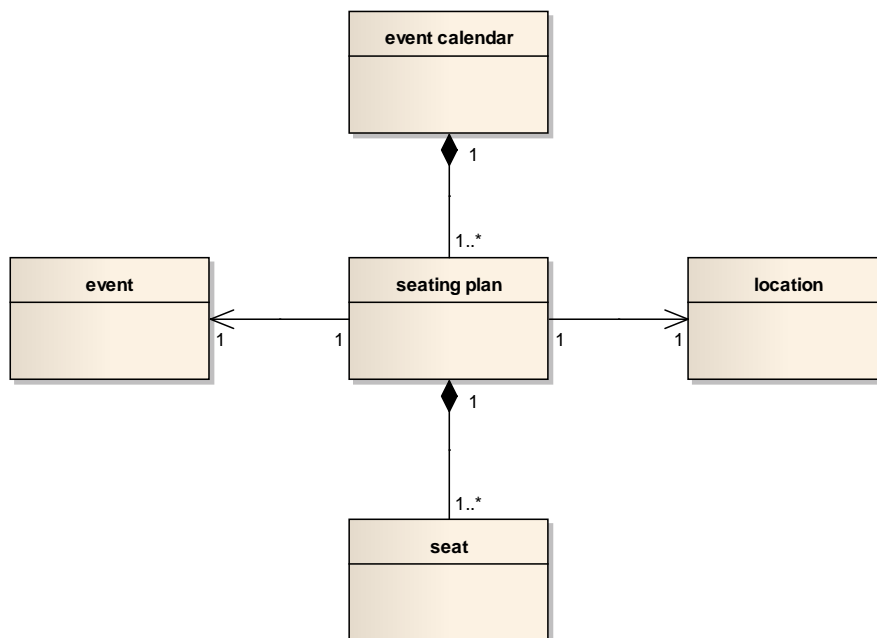


Fig. 51: Example Class Diagram

Chapter Review

Please find the correct answers:

1. A class in UML is shown as

- [a] a rectangle with three areas: Class Name, Methods, States
- [b] a three-tone rectangle with Class Name, Parameters, Methods
- [c] a rectangle with three areas: Class Name, Attributes, Methods

2. The symbol placed in front of an attribute

- [a] “#” indicates attributes which are outwardly visible as properties
- [b] “+” means general visibility (also outside of the class)
- [c] “-“ indicates attributes with simple data types

3. A domain model is

- [a] the depiction of reality in the analysis phase
- [b] a model based on attributes and methods of classes
- [c] a basis for the creation of source code

4. In a composition

- [a] the classes are linked together as individual parts
- [b] one class is an inseparable component of the others
- [c] one class attribute and methods are inherited from the others

5. In Generalization

- [a] the arrow shows the connection to the special class
- [b] the special class methods and attributes are inherited from the general class
- [c] equivalent attributes are passed between classes and filled.

Correct answers: 1c, 2b, 3a, 4b, 5b

Package Diagram

A Package is a logical aggregation of model elements of any type, and with which the entire model is organized into smaller, more manageable units.

A package defines a namespace; i.e. the names of the elements contained within a package must be clear. Every model element can be referenced in other packages, but can belong to only one (home) package. Packages can contain various model elements, such as classes and use cases. They can be hierarchically organized; i.e. for their part contain packages again.

A model element such as a class can be used in various packages; however, every class has its home package. In all other packages, it is simply referred to by its qualified name

PackageName::ClassName

Dependencies thereby arise between the packages; i.e., a package utilises the classes of another package.

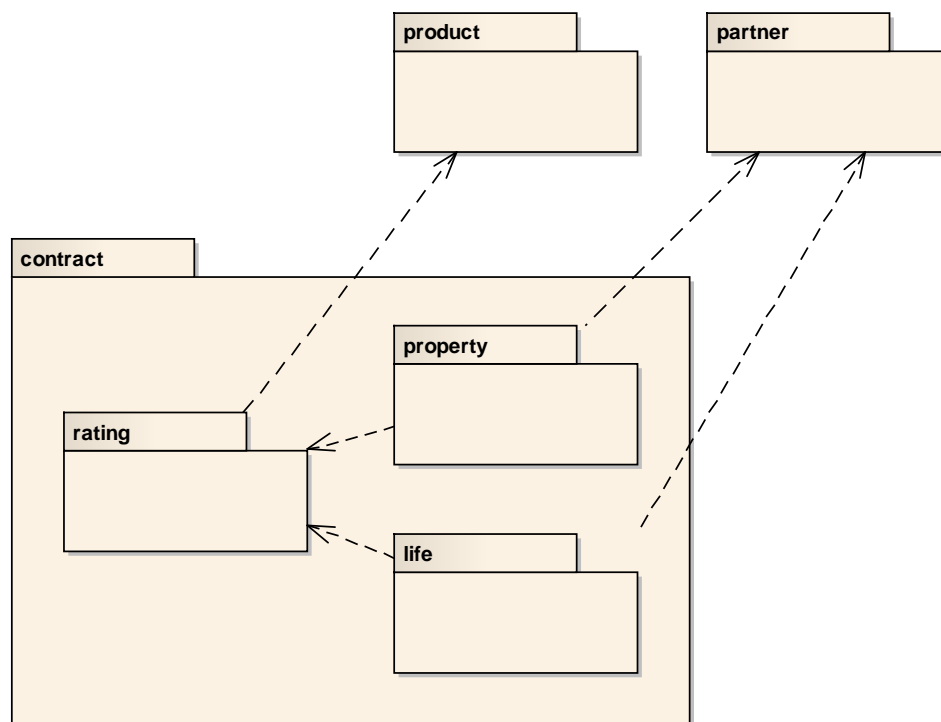


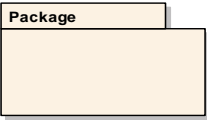
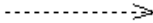
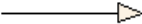

Fig. 52: Example Package Diagram

A package is shown in the form of a file register card. The name of the package is in this symbol. If modeling elements are shown within this symbol, the name is on the register tab; otherwise, within the large rectangle. Stereotypes can be noted above the package name.

If a task is to be distributed to several developer groups, a division into packages can be used and each group given such a package for handling. To ensure problem-free combination of the packages later on, an interface is stipulated whereby at first only the name of the interface is necessary. The functions contained within the package are called up via the interface. In this way, the package can be appropriately tested and subsequent cooperation can be guaranteed with additional packages. Guaranteed means here that the defined resulting values for the provided input parameter are delivered for the functions implemented in the package.

The interaction of system packages is shown in a package diagram in which one can also model the internal structure of a package.

Graphical Elements

Name/Element	Use
<p data-bbox="312 293 408 322">Package</p> 	<p data-bbox="499 293 1393 412">This symbol for a package is reminiscent of a file register card. The package name is shown within this symbol. Should you wish to model inter-package communication, the package name can be placed in the rectangle attached to the top.</p>
<p data-bbox="264 483 454 512">Communication</p> 	<p data-bbox="499 483 1393 568">Communication among the packages is shown with an arrow which runs along a broken line. The arrow starts from the package from which communication predominantly originates.</p>
<p data-bbox="272 591 446 620">Generalisation</p> 	<p data-bbox="499 591 1393 676">Should one package inherit from another package, the symbol for generalization is applied. The involved packages are linked by an arrow, whereby the arrow points to the package from which is inherited.</p>
<p data-bbox="260 698 459 728">Include (Nesting)</p> 	<p data-bbox="499 698 1393 817">Using this symbol, package arrangement can be modeled from the superordinate package, whereby the "Include" symbol is placed on the symbol of the superordinate package. The subordinate packages are linked with this symbol by a solid line.</p>

Chapter Review

Please identify the correct answers:

1. In a package

[a] one finds only the collected classes and objects of a component

[b] no additional sub-packages may occur

[c] any number of elements may be combined

2. A Namespace

[a] is the area in which all elements must have a distinct name

[b] is noted with vertical channels

[c] is separated by a colon after the object name

Correct Answers: 3c, 4a

Interaction Diagrams

An interaction describes an array of messages which are exchanged by a selected number of participants in a chronologically-limited circumstance.

UML 2.0 specifies three different diagrams for the illustration of interactions:

- Sequence Diagrams emphasize the order of message exchange
- Communication Diagrams emphasize the relationships among participants
- Timing Diagrams emphasize the state change of participants against time and the exchanged messages

Sequence Diagram

A Sequence Diagram is primarily concerned with the chronological progression of messages. The messaging sequence corresponds to its horizontal position in the diagram. When an object is created, and when and to what object information is sent, are all determined here.

The participating objects are represented by a rectangle and a dashed vertical line. Both together are called a Lifeline. Messages are shown using arrows between the Lifelines. Time progresses from top to bottom. The chronological progress of messages is thereby highlighted.

The sequence diagram in the following illustration shows an interaction among three objects. It is important that the entire diagram represents an interaction, and that an interaction is not only a single message exchange.

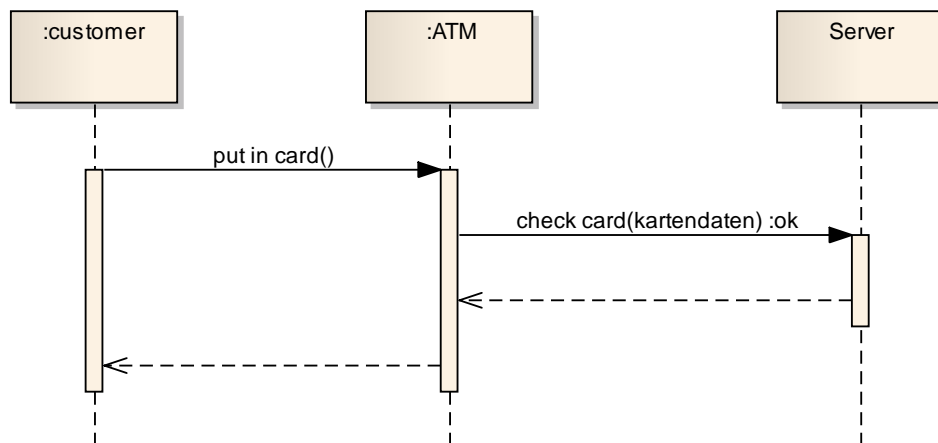


Fig. 53: Simple example of a sequence diagram

In the heading of the Lifeline is the (optional) element name with the associated class in the usual declaration notation: name : type.

ExecutionOccurrence

When messages are exchanged between Lifelines, a Behavior must also be implemented in the associated elements. This is shown by the elongated rectangle on the Lifeline. These rectangles represent the so-called ExecutionOccurrence. Start and End of ExecutionOccurrence are defined via the so-called Event Occurrence. Put more simply, the sending and receiving of messages determines the start and end of the ExecutionOccurrence.

Message Types

The transfer of a message is noted using arrows. Labeling of messages is carried out using the names of the corresponding operations. UML recognizes various types of messages which are demonstrated using various kinds of arrow notation. In the following illustration, the various message types and corresponding notation forms are shown.

- *Synchronous Messages* are represented by filled arrowheads. Synchronous means that the caller waits until the called behavior has ended. The Reply Message to a synchronous call is represented by a dashed line and open arrow point.
- *Asynchronous Messages* have an open arrow point. Asynchronous means that the caller does not wait, but rather proceeds immediately after the call. There is accordingly no answer arrow to asynchronous calls.
- *Lost Messages* have an open arrow point in the direction of a filled circle. The circle is not linked to a Lifeline. When a message is lost the sender is recognized, but not the receiver.
- *Found Messages* have an open arrow point. The line emanates from a filled circle. The circle is not linked to a Lifeline. When a message is found the receiver is recognized, but not the sender.
- A message which creates a new Element is represented by a line and open arrow point. The Lifeline which belongs to the Element starts first on this place in the diagram; e.g., the arrow points to the Lifeline header.

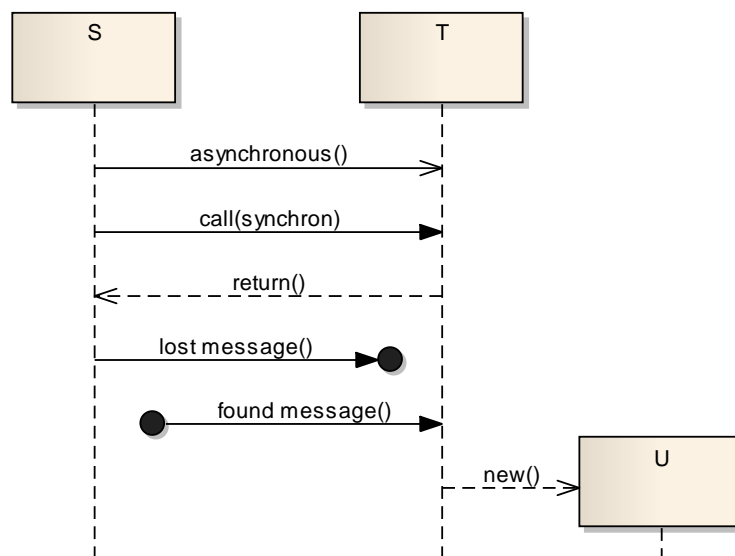


Fig. 54: Notation Forms of the various Message Types

Repeated message sending is modeled by adding the * symbol, in which case the message has the * symbol placed in front.

The message is noted on the message arrow with the following syntax:
[attribute =] name [(arguments)] [: return value]


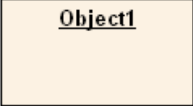
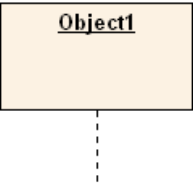

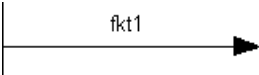
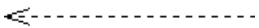
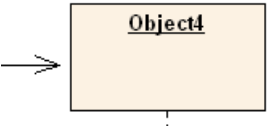

Whereby

- “attribute” can be a local variable of the interaction, or an Element of a Lifeline. The attribute allocation is only used in synchronous messages with return value.
- “name” is the name of the message to be called, or the name of the signal to be sent. The sending of a signal is always as asynchronous character.
- “arguments” is a comma-separated list of parameter values transferred to the message.

If an object is created via the setting of a message (e.g. by calling the method “new”), then the Lifeline of this object begins at this position. The destruction of an object is represented by a cross on the Lifeline.

Symbols

The following table contains the sequence diagram symbols.

Symbol/Name	Use
System Border 	The System Border isolates the concerned part of the program from the rest of the program. It usually serves as the start point of the triggering method call. Program flow is not always triggered by an object outside of the concerned area, so that in this case no system limit must be set.
Object 	An object is shown by a rectangle containing the name. Underlining of the name may be omitted so that no confusion with the class name can occur. Classes are not displayed in this diagram. Objects are shown along the upper sheet edge.
Lifeline 	Every object is on a vertical line - the Lifeline. An object's lifeline grows in the direction of the lower sheet edge. For objects which already exist at the start of the program section, the object symbols are drawn on the upper sheet edge. For objects which are re-created within the program section, the symbol is drawn at the level of the method call in the course of which the object was created.
Object Activity 	If an object is involved in a method call, it is active. The Lifeline thickens. If an object calls its own method, the Lifeline thickens again. These activities are not always drawn.
Method Calls 	When an object calls a method of another object, this is symbolized by a continuous arrow which points to the object from which the method was called. The method name is placed on this symbol. This name can be added to the parameters list in parentheses.
Return of a Method 	In principle, only the method calls are shown in the sequence diagram. Should you nevertheless wish to plot method returns, this can be done with an arrow and a dashed line.
Object Creation 	If a method creates an object, the method's arrow ends on the object's rectangular symbol. The Lifeline begins at this symbol.
Object Destruction 	If an object is destroyed when a method is called, the object's Lifeline ends with a cross below the method call symbol.

Example

A theatre's ticket system allows tickets to be sold out of the seating plan on an Internet website. The seating plan manages the seats of a given event.

When a seat is selected by a user, the concerned object `:seat` calls the "buy" method of the `:"order"` object and transfers a reference to itself in the parameter. The order class object calls the method "isFree(seat)" of the Seating Plan class to check whether the seat transferred into the parameter is free. If the seat is still free, the Seating Plan object calls its own method, "reserve". The seat is thereby reserved for the time being.

After this has taken place, the invoice for the seat is created. The seating plan object then calls the "book" method with the chosen seat as parameter. The "book" method belongs to the object `:order`. This represents a list of the invoice line items - which is not modeled here. After the invoice

line items have been compiled by the object :sales, it calls the “createInvoice” method of the object :order. To notify the Internet ticket system user of the success of his purchase, the Order object calls the method “confirmed” of the class Seat. The visitor confirms the order, and the method “confirmed” returns with the value “true”. The order is completed.

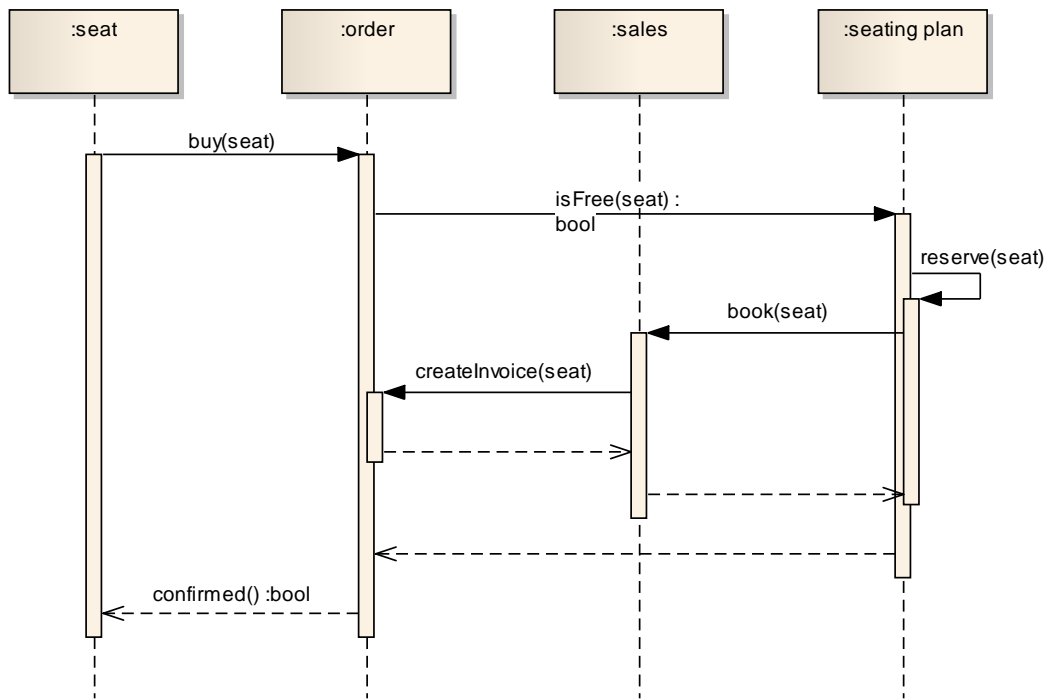


Fig. 55: Example of a Sequence Diagram

Chapter Review

Please identify the correct answers:

1. Interaction diagrams describe

- [a] messages which exchange objects within a concerned time frame
- [b] the communication between attributes of objects
- [c] interfaces to users and other systems

2. Execution Occurrence is shown by

- [a] a dashed line under an object
- [b] a thickening of an object's Lifeline
- [c] an arrow with an open point which connects two object Lifelines

3. A message for which the caller does not await an answer

- [a] is called an "asynchronous" message
- [b] is not modeled in the UML diagram
- [c] is marked by an "X" on the object's Lifeline

4. New objects in a Sequence Diagram are

- [a] modeled only on the upper page edge, and live during the entire interaction
- [b] are modeled on the position of the first method call
- [c] are modeled with the help of the "lost message"

Correct Answers: 1a, 2b, 3a, 4b

Communication Diagram

The Communication Diagram corresponds to the UML 1.x Collaboration Diagram. It has been renamed to avoid the confusion caused by the term “Collaboration”, as UML also has the modeling element “Collaboration” which has nothing to do with a Collaboration (Communication) Diagram.

The Communication Diagram is a different approach to displaying the circumstances of a sequence diagram. This diagram gives special attention to the cooperation of the interconnected objects. Selected messages are used with which the chronological communication sequence between the objects takes place. It thereby compiles the conclusions of the sequence diagram in a more compact form.

In the following illustration, one can clearly see that the communication diagram emphasizes the relationships between the involved parties, and not the chronological progression of message exchange like the sequence diagram.



Fig. 56: Example of the “Identify Authorization” Communication Diagram

Graphical representation is a rectangle which contains the object name and the respective class. A colon separates both names. The objects are linked by association lines. A small arrow shows each message direction from sender to receiver. When arguments are transferred with the message, these are executed. Possible return values can also be output in the form:

answer := Messagename (list of Parameters)

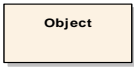

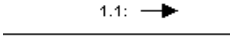
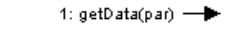
In order to model the chronological sequence, the messages are given numbers. One message can trigger further messages. These receive sub-numbers of the triggering message (e.g. 1.2).

If a message is repeatedly triggered, this iteration can be modeled using a * character in front of the message name.

Objects which are created within the illustrated scenario can be indicated using the stereotype *new*. Objects which are destroyed within the illustrated scenario are indicated using the stereotype *destroy*. Objects which are created and destroyed within the scenario receive the stereotype *transient*.

Symbols

The following table contains the Communication Diagram symbols.

Symbol/Name	Use
Object 	This rectangle is the symbol of an object and contains the object names. No confusion with classes may occur, so the name must not be underlined.
Communication 	If two objects communicate with one another via a method call, this link is shown with a solid line connecting both objects.
Communication Direction 	In addition to the connecting line, after the method name an arrow shows to which object the method belongs. The arrow points to this object.
Communication Line Labeling 	This line is labeled with the method name and the parameter list. Numbers placed in front of the method name indicate the chronological order of method calls, separated by a colon. You can enter the method numbers in a condition which are, as a precondition, already being processed. The * symbol in front of the method name indicates a repeat of that method. The condition for repetition can be entered after the * symbol.

Example

This example models the purchase of a ticket via the Internet. The Internet customer selects a seat on a theatre's website, thereby calling the "select" method. The interaction is started. The object `:seat` calls the "buy" method of the Order object and transfers a reference to itself in the parameter. The `:order` object calls the method `isFree` of the object `:seating plan`. On the one hand, this method calls the "reserve" method, and on the other hand the "book" method. They therefore differ only in their sub number. The Seating Plan object calls the method "book" of the `:sales` object. This object calls the method "createInvoice" of the object Order and transfers the invoice line item of the booked seat. After the invoice has been created by the object `:order`, the Internet customer is informed of the successful booking, whereby the method "confirmed" is called which queries and returns the customer's confirmation.

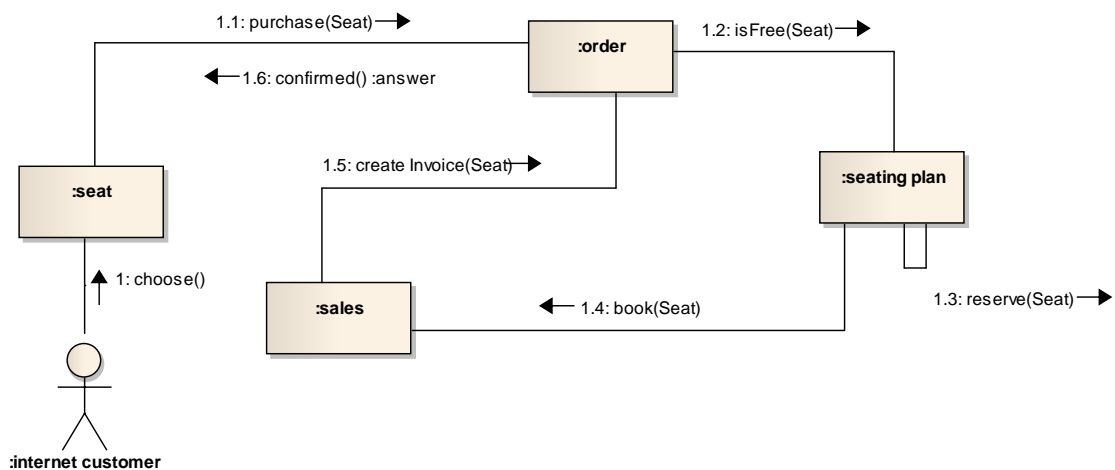


Fig. 57: Example of the "Ticket Purchase via Internet" Communication Diagram

Sequence Diagrams vs. Communication Diagrams

Sequence and Communication Diagrams are very similar and can also be merged into several UML Tools. However, due to the greater expressiveness of sequence diagrams in UML 2.1 it is no longer possible to substitute every sequence diagram with a communication diagram. The focus of the sequence diagram is on the chronological aspect, while for the communication diagram focus is on the relationships between the objects. The greatest advantage of the sequence diagram - and at the same time greatest disadvantage of the communication diagram - is the clearly visible chronological sequence which can principally be visualized in communication diagrams with a numbering scheme, but much less visibly. In this case the order of execution must not be set right away, making the creation of a sequence diagram somewhat tricky at times. On the other hand, in a comprehensive communication diagram the order is not easily readable. The reader must use the Search function to find out whether 1.1 2, 1.2 or even 1.1.1 follows.

Chapter Review

Please identify the correct answers:

1. Sequence and Communication Diagrams

- [a] have nothing to do with each other
- [b] exchange messages via “lost” and “found” messages
- [c] are similar and can be substituted in some cases using various tools

2. The chronological order in a communication diagram is

- [a] visible via a numbering scheme
- [b] visible via the names of the method calls
- [c] more visible than in a sequence diagram

3. The direction of communication flow between objects in a communication diagram

- [a] is modelled using an open arrow point on the association
- [b] is labelled with dashed (broken) and solid (unbroken) messages
- [c] is symbolised by a closed arrow point next to the method name

4. In interaction diagrams, object names are

- [a] always underlined when shown with classes due to danger of confusion
- [b] often shown not underlined as there is no danger of confusing them with classes
- [c] modeled with no underline but therefore always with package and class name in front

Correct Answers: 1c, 2a, 3c, 4b

Interaction Overview Diagram

The Interaction Overview Diagram is new in UML 2.0/2.1. It merely represents a mix of activity and sequence diagrams, whereby activity blocks can be mixed into a sequence diagram, and vice versa. Since it contains no other innovations, no further details will be explored here.

The following example shows the “Execute Bank Transfer” process, whereby the main diagram here is an activity diagram and an action (save order) is more precisely described using a sequence diagram.

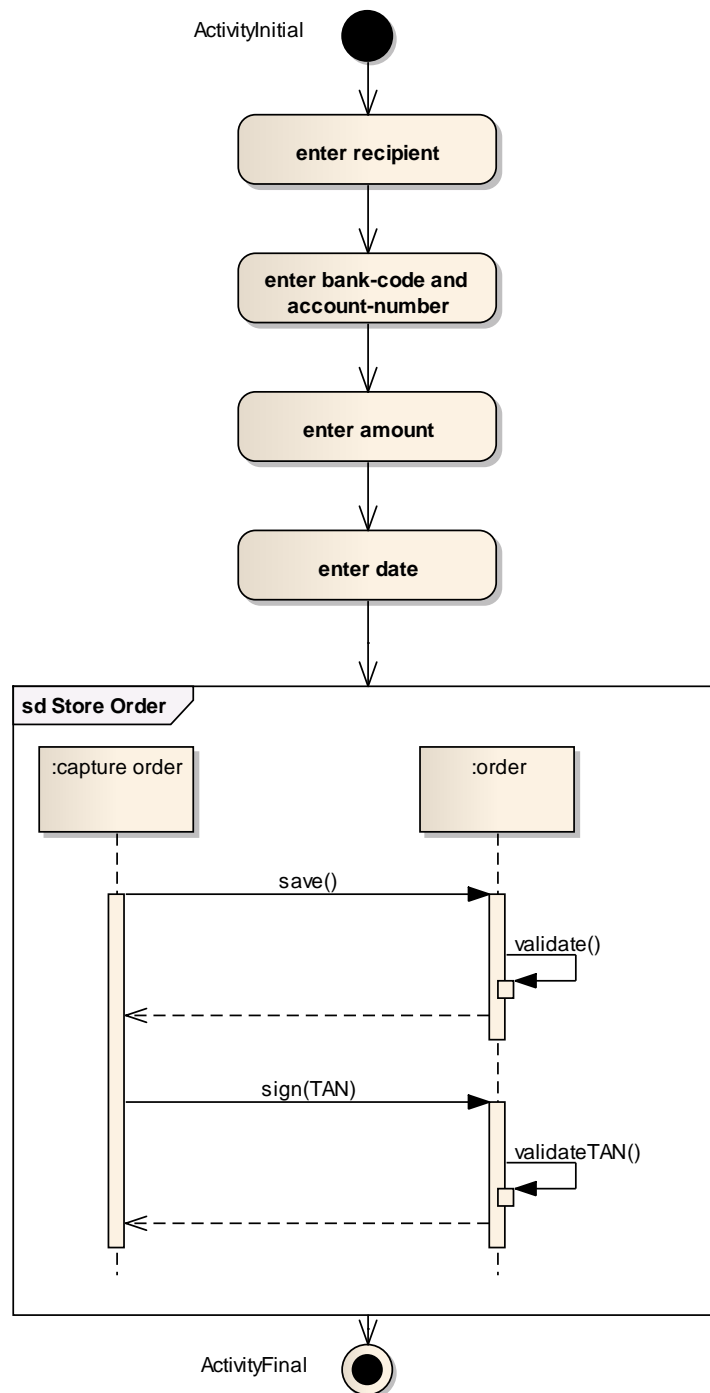


Fig. 58: Example Interaction Overview Diagram

Component Diagram

Component diagrams show the interaction of component relationships. A component is an executable and exchangeable software unit with defined interfaces and individual identities. A component is, like a class, instanceable and capsules complex functionality.

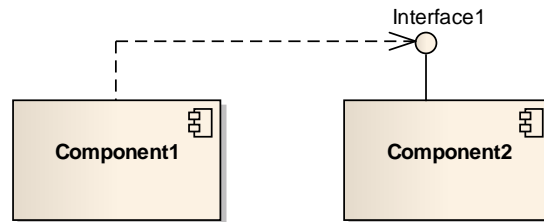
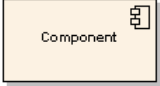
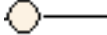
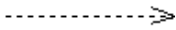


Fig. 59: Notation of Component Diagram

The component *Component2* provides functionality via its interface. *Component1* communicates over the interface *Interface1* provided by *Component2* and utilises its functionality.

Symbols

The following table contains the component diagram symbols.

Name/Symbol	Use
Components 	The component symbol is an arrangement of three rectangles. Two rectangles are drawn on the left side of the third, larger rectangle - where the name of the component is located. Components completely fulfill the tasks of an autonomous area for which it was developed.
Interface 	A component can provide an interface. This is where administered access to component functionality takes place. Interfaces make these elements exchangeable. An interface is shown with a circle. This circle is linked to the class, the package or the component by a solid line. The name of the interface is given next to the symbol.
Dependency 	This arrow on a dashed line symbolizes access to the interface. The arrow points to the interface symbol circle. It can also point directly to a component when access does not take place via the interface.

Example

In this example, four components are linked to an application to show database data in a window. The database access component is responsible for the establishment, management and interruption of database access. It provides methods via an interface over which the data container component can request data from the database.

The data container component manages the requested data sets. At the same time, a transaction is started for each data transfer via the transfer management component. The component for graphical representation calls the data sets via the data container component interface and shows these on the screen.

Should the user alter the data, the component transfers these changes to the data container component. The altered data are written by these in the database via the connection component and the new data are verified over the Data Transfer component, then the started transaction ends.

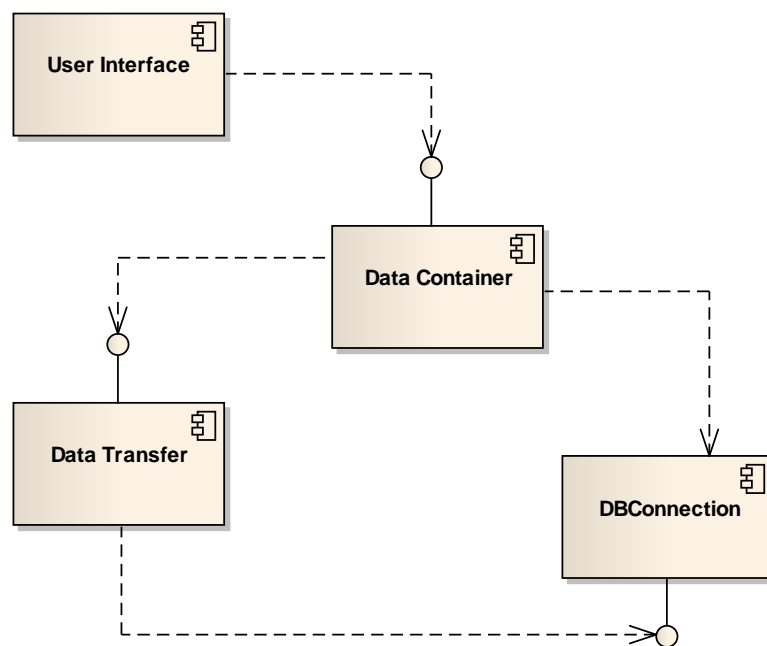


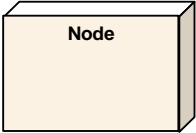
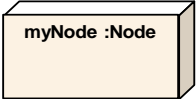



Fig. 60: Example Component Diagram

Deployment Diagram

Dependencies between nodes, the nodes themselves, their contents and communication are shown in this diagram. A node is a physical unit which possesses storage capacity and computing power (such as a PC) on which further components are run. These nodes are drawn as blocks. Components or processes can be drawn within these blocks. Images which graphically represent these nodes may also be used instead of blocks. Distribution diagrams show which components and objects run on which nodes (computers, processes), i.e. how these are configured and which communication relationships exist.

Symbols

The following table contains the Deployment Diagram symbols.

Symbol/Name	Use
<p>Node</p> 	<p>A Node is represented by a block. The label placed within the block contains the name of the node, e.g. the name of a computer, a client or a process name.</p>
<p>Subsystem</p> 	<p>If an application is carried out in a node's memory storage, this application is modeled as a node instance. To differentiate with the node symbol, the label within the block is underlined. This symbol is inserted into the symbol for the node on which the sub-process is running.</p>
<p>Component</p> 	<p>A Component is in scope smaller than a subsystem but assumes application tasks for a specified application range. The Component symbol is inserted into the symbol of the node on which the component is installed.</p>
<p>Component Instance</p> 	<p>Component Instance is a general term for specific objects, the classes of which belong to the component's data structure and are instanced by it. This symbol does not differ from the component symbol. Differentiation to the component symbol is achieved in that the name of the instance is underlined. The symbol is inserted into the component symbol's instance which it represents.</p>
<p>Association</p> 	<p>Relationships between nodes are shown with a solid line which links the nodes.</p>

Example

A cinema's ticket data can be accessed via a ticketing system using various applications. The data are on a central server which is networked with other PC's and connected to the Internet. A component for the administration of the database is installed on the server. The database server Interbase is set up on this server.

- Box office sales are processed via a PC in the box office area. In order that this PC can access the data quickly enough, a client application is developed which is especially adapted to the requirements of box office sales.
- The system administrator has a PC for its tasks, e.g. in bookkeeping. He also has a client application which enables him to create and evaluate events.
- The pre-sales department is driven with its own client application which is connected to the server via the Internet.
- The Internet customer calls up data from the ticket system from a PC. A ticket system client application tailored to the needs of this customer is required for this procedure.

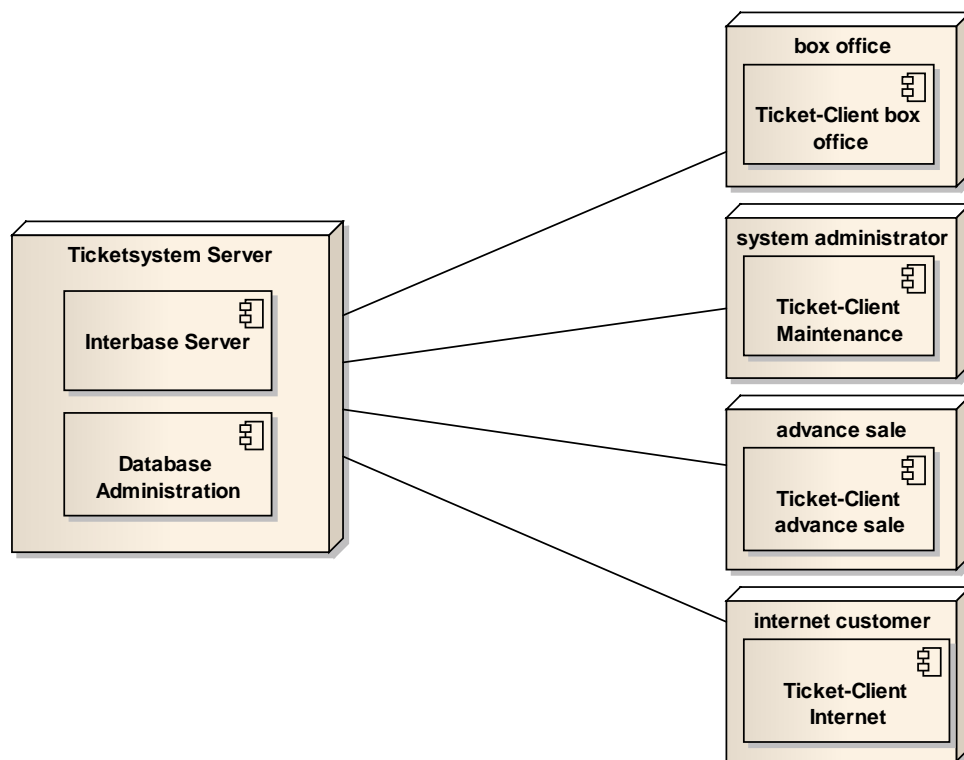


Fig. 61: Example Deployment Diagram

Chapter Review

Please identify the correct answers:

1. The Interaction Overview Diagram

- [a] is a mix of communication and sequence diagrams
- [b] represents nodes and the subsystems found on them
- [c] is a mix of activity and sequence diagrams

2. To model capsuled software units with defined functionality,

- [a] the instanced classes of the class diagram can be used
- [b] one utilizes components and interfaces in the component diagram
- [c] the best way is to implement packages and associations

3. An interface provided by a component

- [a] allows access to the functionality of the component
- [b] is symbolized by a large rectangle with two small rectangles
- [c] helps to keep system components unique by way of their names

4. A node in a Deployment diagram is

- [a] a physical unit which possesses storage capacity and computing power
- [b] a point at which information converges and is stored
- [c] a unit in which a finished software product can be subdivided

Correct Answers: 1c, 2b, 3a, 4a

Timing Diagram

The Timing diagram is primarily implemented in hardware-oriented programming or in chronologically-critical organization projects to analyze processes for optimal flow using a time line.

The following illustration shows a time diagram. Here, emphasis is placed on the state of those involved with respect to a timeline.

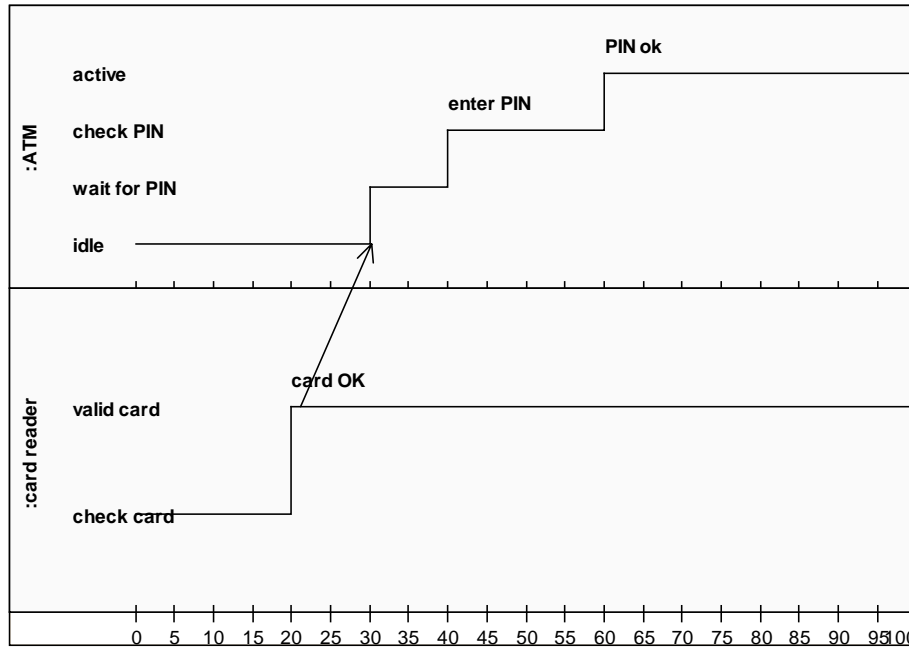


Fig. 62: Example Time Diagram

Hint: In opposition to the timing diagram a sequence diagram has no linear time scale. But you may use timing-arrows there expressing desired timing relations. Due to this you may prefer to use sequence diagrams.

Composite Structure Diagram

The internal structure of a class can be described with the Composition Structure Diagram. This diagram is new in UML 2.0/2.1. At first glance, the diagram can be easily mistaken for a communication diagram. The following illustrations show a composition structure diagram and an associated class diagram with compositions. The assertions of both diagrams are equal.

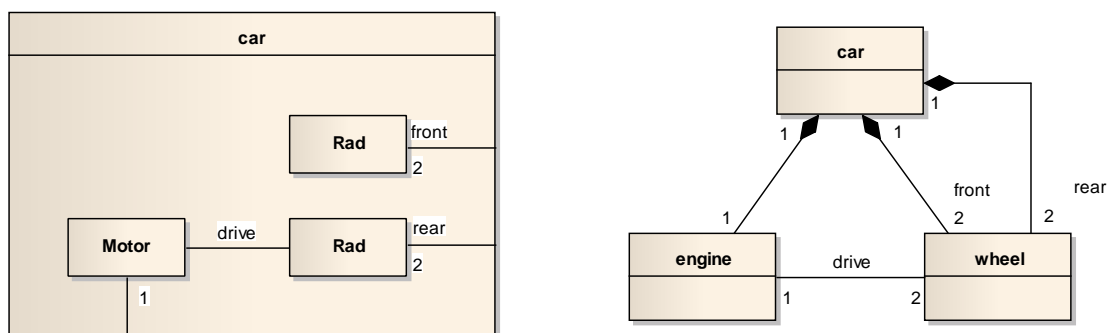


Fig. 63: Composition Structure Diagram and equivalent Class Diagram

Object Diagram

The Object diagram holds a certain similarity to the class diagram but with the decisive difference that, here, only instances and no classes are displayed. It shows a particular detail of the program during runtime. The individual objects are shown, along with the links and also the multiplicities. For example, this diagram type is implemented during readjustment of errors in the running system. When a particular software behavior occurs under specific conditions, this can be described using the Object diagram by showing the relevant attributes for this circumstance and their values in the objects.

The following example shows a class diagram to the left and the corresponding object diagram to the right.

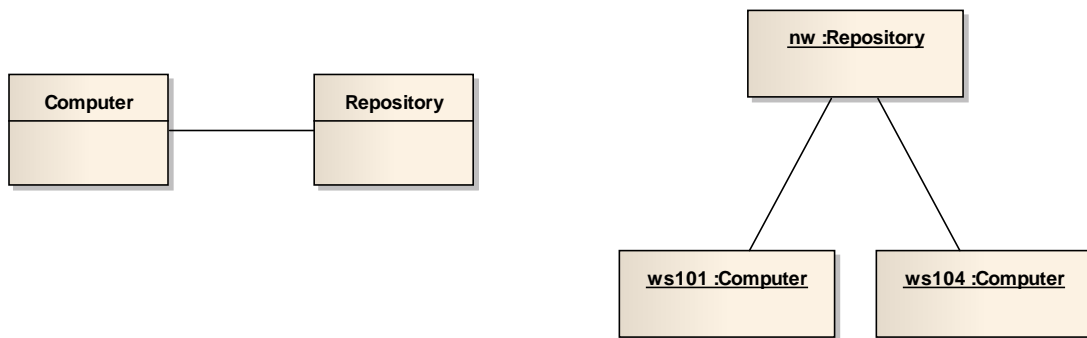


Fig. 64: Example of Object Diagram (right) and corresponding Class Diagram

Chapter Review

Please identify the correct answers:

1. The Timing Diagram is not

- [a] implemented in hardware-oriented programming
- [b] used for the analysis of time-critical organization projects
- [c] to be referred to as a representation of a user's chronological sequence of activities

2. The relevant relationship which is cancelled using the Composition Structure diagram

- [a] is called Composition and is found in the Class diagram
- [b] is called Collaboration and is found in the Component diagram
- [c] is called Composition and is found in the Structure diagram

3. An Object diagram helps to

- [a] find errors in a system which occur during runtime
- [b] illustrate the messages sent between objects
- [c] achieve an overview of the system environment

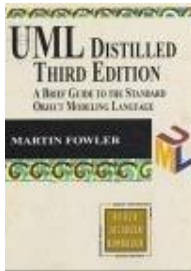
Correct Answers: 1c, 2a, 3a

Images

Fig. 1: Forward, Reverse and Round-Trip Engineering.....	5
Fig. 2: Historical Development of UML.....	7
Fig. 3: Diagram Frame Example	8
Fig. 4: Overview of UML Diagrams	8
Fig. 5: Use Case Diagram.....	11
Fig. 6: Notation of Actors	12
Fig. 7: Notation of use cases	12
Fig. 8 System	12
Fig. 9 Multiplicity and active/passive actors	13
Fig. 10: Example of «include» relationship	14
Fig. 11: Example «extend» relationship.....	14
Fig. 12: Example «extend» with extension points and condition.....	15
Fig. 13: Generalisation of Use Cases	15
Fig. 14: Generalisation of Actors.....	15
Fig. 15: Notes in Diagrams	16
Fig. 16: Example of a Use Case Diagram.....	17
Fig. 17: Example of an Activity, “Production of Sixpacks”	19
Fig. 18: Control Flow / Object Flow	20
Fig. 19: Parallelization and Junction – implicit vs. explicit	21
Fig. 20: Merging.....	21
Fig. 21: Synchronization = Join	22
Fig. 22: JoinSpec	22
Fig. 23: Calling an Activity by an Action.....	23
Fig. 24: Structured (composite) activities.....	23
Fig. 25: Send / Receive	24
Fig. 26: Interrupt Region	24
Fig. 27: Example of Activity: Prepare for Party.....	28
Fig. 28: Example State Machine Diagram	31
Fig. 29: Example State Machine Diagram “Automatic Teller Start-up”	33
Fig. 30: Class Example.....	35
Fig. 31: Example Stereotypes.....	36
Fig. 32: Parameterized Class	36
Fig. 33: Object Example.....	36
Fig. 34: Association and Composition with all properties	37
Fig. 35: Associations	38
Fig. 36: Multiplicity vs. Cardinality	38
Fig. 37: Association Class	39
Fig. 38: Association Node.....	39
Fig. 39: Aggregation notation.....	40
Fig. 40: Example Aggregation.....	40
Fig. 41: Aggregation and Composition	40
Fig. 42: Example Composition	41
Fig. 43: Example Aggregation and Composition	41
Fig. 44: Example Inheritance.....	42
Fig. 45: Example interface	44
Fig. 46: Notation for Available/Provided Interface	44
Fig. 47: Notation for Requested/Used Interface.....	44
Fig. 48: Notation Options for Interfaces (Usage and Realize)	45
Fig. 49: Extension of Interfaces	45
Fig. 50: Example of Implementation of Interfaces.....	46
Fig. 51: Example Class Diagram	47
Fig. 52: Example Package Diagram	49
Fig. 53: Simple example of a sequence diagram	52
Fig. 54: Notation Forms of the various Message Types	53
Fig. 55: Example of a Sequence Diagram	55
Fig. 56: Example of the “Identify Authorization” Communication Diagram.....	57
Fig. 57: Example of the “Ticket Purchase via Internet” Communication Diagram	58
Fig. 58: Example Interaction Overview Diagram.....	61
Fig. 59: Notation of Component Diagram.....	62
Fig. 60: Example Component Diagram	63

Fig. 61: Example Deployment Diagram	65
Fig. 62: Example Time Diagram	67
Fig. 63: Composition Structure Diagram and equivalent Class Diagram	67
Fig. 64: Example of Object Diagram (right) and corresponding Class Diagram	68

Recommended Additional Literature



UML Distilled. Third Edition
by Martin Fowler

The classic UML book.
Short, clear presentation of the most important features in UML

<todo> Oliver Alt
SysML (falls auch
engl)



Real Time UML
by Bruce Powell Douglass

The classic on embedded and real-time developers!



UML@Work - 3rd Edition
by Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger

For UML beginners as well as advanced UML users. Language concepts and notation of diagram types for structure and behavioural modelling based on a continuous example.

Index

- Abstract Class 35
- Abstraction relationship 43
- Activity 19
- Activity Diagram 19
- Activity Final 26
- Actor 11
- Aggregation 40
- Association 37
- Association Class 39
- association node 39
- Asynchronous Messages* 53
- Attribute 37
- Available Interface 44
- Booch 6
- call 42
- Call Behaviour Action* 22
- Calling an Activity by an Action 23
- Cardinality 38
- Class 35
- Class Diagram 35
- Collaboration Diagram 57
- Communication Diagram 57
- Component 64
- Component Diagram 62
- composite Element* 23
- Composite Structure Diagram 67
- Composition 40
- create 42
- Creation of Objects 54
- dependency 42
- Deployment Diagram 64
- derive 42
- destroying an object 54
- Diagram Implementation 9
- element linking (composite) 23
- Execution Occurrence 52
- Extend Relationship 14
- Flow Final 26
- Found Message* 53
- Generalisation 15, 41
- Harel, David 31
- Include 50
- Include Relationship 13
- instantiate 42
- Interaction Diagrams 52
- Interaction Overview Diagram 61
- Interface 43
- Introduction to UML 5
- Jacobson 6
- JoinSpec 22
- Junction 21
- Lifeline 52, 54
- linked (composite) element 23
- Lost Message* 53
- Message 52
- Message Types 52
- Method 37
- Multiplicity 38
- n-ary Association 39
- Nesting 50
- Node 64
- Object 36
- Object Activity 54
- Object Creation 54
- Object Destruction 54
- Object Diagram 68
- Operation 37
- Package Diagram 49
- Parameterized Classe 36
- Partition* 23
- permission 43
- permit 42
- provided interface 44
- Realization* relationship 43
- realize 42
- refine 42
- Relationship 12
- requested interface 44
- Responsibility Zones 23
- Return of a Method 54
- Rumbaugh 6
- Scope 35
- Sequence Diagram 52
- Specialisation 15, 41
- State 32
- State Machine Diagram 31
- Stereotype 36
- structuring elements 23
- Subactivities 25
- Substitution* relationship 43
- Swimlane* 23
- Swimlanes 23
- Synchronisation 22
- Synchronisation with JoinSpec* 22
- Synchronous Message* 53
- Timing Diagram 67
- trace 43
- Transition 32
- UML Diagram Types** 8
- use 43
- Use Case 12
- Use Case diagrams 11