VFHVWS

//vfhvws.oncampus.de

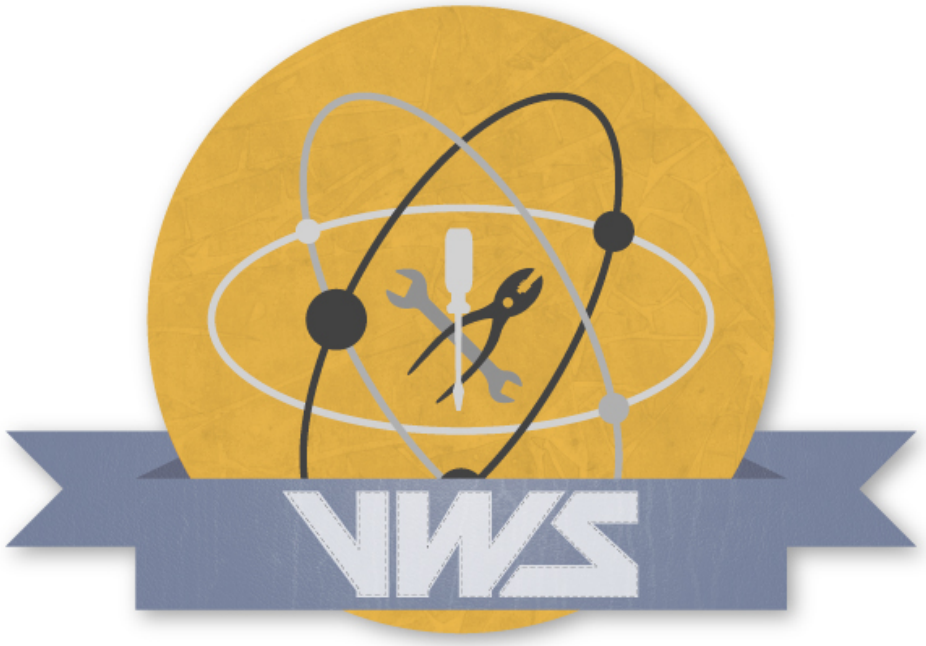
Stand 02.11.2017 10:31

Inhalt

VFHVWS

Online Studienmodul

Verfahren und Werkzeuge moderner Softwareentwicklung



Lerneinheiten

Über den Autor

Prof. Dr. Stefan Edlich

Hochschule:
Beuth Hochschule für Technik Berlin
Tätigkeitsbereich:
NoSQL - Big Data / Data Science - Software Engineering
Kontakt:
FB VI / Labor Online Learning
Luxemburger Str. 10
13353 Berlin

1 UML - Unified Modeling Language

Autor: Prof. Dr. S. Edlich

1.1 Overview and Learning Objectives UML

Some of you may have already had this learning unit in your bachelor's degree (e. g. in software engineering). This session is intended for you as a repetition and introduction to the UML task you have to submit. For all others who have not yet got to know UML during their studies, this course is an important basis which you will need, among other things, to solve the UML submission task.

Lernziele

Learning Goals The aim of this session is to familiarize you with UML. In addition to the introduction, static structure diagrams and dynamic diagrams (behavioral diagrams) are presented.

You should be able to judge the appropriate use of UML and apply it to your own project. The various elements of UML are to be used and the advantages and disadvantages of this representation form are to be recognized and described. The critical use of this industrial language should definitely be mastered.

As a result of the practical exercise, you will be able to assess which UML diagrams you use in which order to achieve your modeling goal.

Gliederung

Outline

- Introduction
- Background and history of UML
- Presentation of static structure diagrams
- Presentation of dynamic diagrams
- Knowledge check and exercises

Zeitumfang

Time Requirements
To work through the learning unit, you will need approx. 120 minutes and for the processing of the exercises approx. 170 minutes.

1.2 Unified Modeling Language

Definition

Unified Modeling Language
The Unified Modeling Language is a set of notation elements that can be used to develop models for software systems. This concerns the analysis, design and in general the presentation and documentation of the software elements or the software behavior.

The UML was developed by the OMG (http://omg.org) and is specified in more than one hundred pages. It has established itself as the most important notation / language for the specification of software systems. One of the most important tasks is to make the design and the associated architectures transparent for other project participants or outsiders. It is important to note that UML does not have to be used dogmatically.

The aim of UML diagrams or descriptions is essentially that other people understand the statement. There is, therefore, no body that decides on the correct and wrong UML. UML itself can be interpreted in many places.

It is certainly better to put energy into good design than 100% correct UML. The use of representations and methods that are not UML are important for this. For example, decision tables are often required for situations. These are not part of the UML, but they are extremely helpful.

Experiment with UML as well as with Non-UML and talk to people who have already had experience in the field. For example, many architects believe that a good design session on a whiteboard without strict UML can do more than just a lot of UML diagrams on paper.

The OMG also manages other interesting standards such as CWM, MOF and the important UML exchange format XMI.

1.2.1 UML Tools

In order to be able to test everything you need a UML tool.

More than one hundred UML tools can be found at http://www.jeckle.de/umltools.htm The German company OOSE also has a good tool page. It will take some time to find' your' tool. Allow time for this! Often, some settings get jammed, and the tools don't show what you want or even generate incorrect Program Code.

Here are just a few tips on which tools have proven themselves here and there:

- **Together** www.borland.com - Flexible and powerful tool
- **Omondo** www.ejb3.org - Good Eclipse integration. Might now have another name
- **Rational Rose** www.ibm.com - Not free of charge, powerful, integrated into the rational Toolsuite, has the reputation to be a bit clumsy.
- **Magic Draw** www.nomagic.com - Feed Required
- **Argo UML** argouml.tigris.org
- **BlueJ** www.blueJ.org - Has emerged as a teaching tool, is closely interwoven with development and can only be used for class diagrams.

- **Fujaba** www.fujaba.de - A well known free tool of the Uni-Paderborn
- **Astah** www.astah.net - From Japan
- **ObjectIF** www.microtool.de from Microtool from Berlin!
- **Innovator** www.mid.de - different versions for different views (Db, architect, developer etc.)
- **Enterprise Architect** www.sparxsystems.com - Very powerful and comparatively inexpensive modeling tool for UML and other languages.

More Recommendations:

- http://www.topcased.org
- http://www.umlet.com

But the Master is:

- http://plantuml.com UML as a textual description! Versionable!
- http://planttext.com for live demos and live generation

1.2.2 Diagram Overview

Structure diagrams and architecture diagrams are also referred to as static diagrams. Behavior and interaction diagrams are often referred to as dynamic diagrams.

Package Diagram	Shows the concrete structure of the software in namespaces or packages with their dependencies.
Class Diagram	Represents the classes of the respective programming language and their relationships.
Object Diagram	Shows objects - i. e. in which state the classes can occur.

Structure Diagrams

Deployment Diagram	Shows where the (mostly hardware) systems and their components are installed in their parts on computer nodes.

Component Diagram	Displays the component structure with the relationships among each other.
Composition Structure Diagram	Shows the components with their specific ex- and imported interfaces.
Subsystem Diagram	Architectural correlations of components.

Architecture Diagrams

Use-Case-Diagram	Presents the actors and their cases of application (quasi their desires).
Activity Diagramm	Shows how the program works by using actions, transitions and branches.
State Diagram	Object states are displayed here. There are transitions between the states. An automat can specify the sequence of transitions even more precisely.

Behavior Diagrams

Sequence - Diagram	Represents how classes or components interact / communicate by means of messages.
Interaction Overview	A combination of activity and sequence diagram.
Communication Diagram	Arranges the exchanged messages graphically to display them appropriately.
Timing- Diagram	Describes the chronological sequence of the object states (often used in telecommunications and control engineering).

Interaction Diagrams

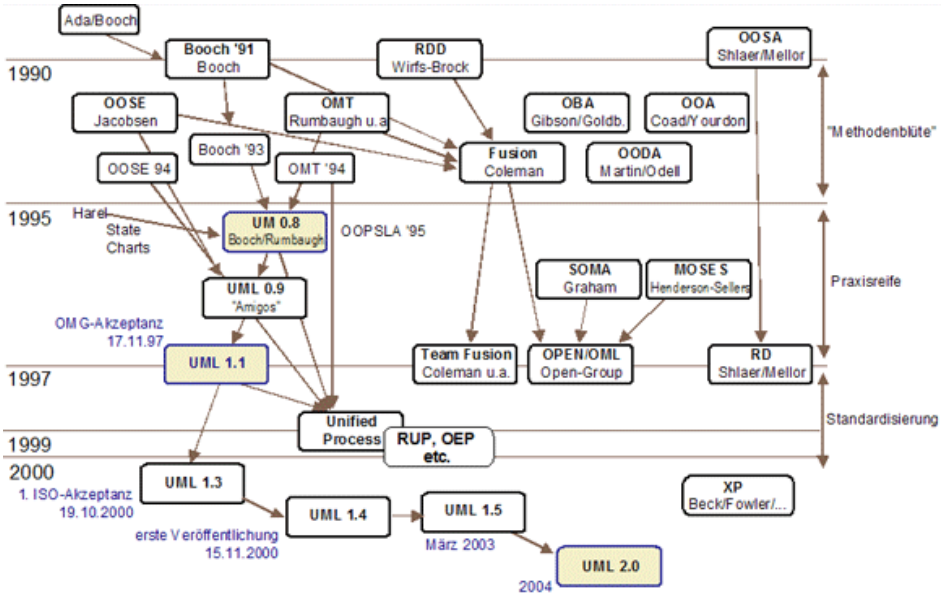
1.3 History of UML

It was not until the 1990s that the work on the subject of object-oriented analysis / design gained momentum. Authors who made a name for themselves here early on were Adele Goldberg, Grady Booch, Peter Coad, Edward Yourdon, James Rumbaugh, Bertrand Meyer and Ivar Jacobsen.

Early on, the methods of James Rumbaugh and Grady Booch became established. However, both competed.
Rumbaughs
method seemed to be better for the analysis and
Boochs
method was probably more advantageous in design. James Rumbaugh was also hired by the Rational Software Cooperation in 1994, where he was also responsible for the further development of the IBM Toolsuite.

United with the forces of Ivar Jacobson,
Rumbaugh
and
Booch
then tried to bridge their differences and create a unified language. This, of course, is also due to pressure from the industry. Ultimately, UML-based tools and process models made it possible to earn money.

Ivar Jacobsens
Objectory was then bought by Rational and the Unified Method (UM) was born from the unification of the methods.



History of the UML

Booch, Rumbaugh
and Jacobsens the three Amigos were called from now on. From the time of their former violent battles for methods, the discussion process and the difficult negotiations, a self-critical statement of
Jacobsens
is known, which is gladly quoted over and over again:

However, the pressure to develop a non-proprietary language based on UM has increased: The Unified Modeling Language (UML). An international consortium developed the UML in 1996 and then handed it over to OMG. This is how the UML 1.1 was created in 1997. It developed in steps to version 2.0, which was the last downloadable version until 2006. At the moment version 2.5 from June 2015 is available in the download area of the OMG.

1.3.1 UML Structure

The latest UML specification can be downloaded from the OMG website (http://www.uml.org) - older versions are also available. It is worthwhile to have a look around on the UML websites. The specification is divided into the following parts:

Vertiefung

UML 2.0

- UML 2.0 Infrastructure Specification: The basic elements of UML are presented here, which are a prerequisite for the following specifications. Elements such as class, association or multiplicity are introduced.
- UML UML 2.0 Superstructure Specification: Here, the actual known elements of UML are defined and presented with use cases and deeper concepts.
- UML 2.0 Object Constraint Language: The last document specifies boundary conditions for some types of graphs. It is used, for example, for class diagrams (chapter 1.5.4) or sequence diagrams (chap. 1.6.4) important. You can specify restrictions with invariants, pre-/post-conditions or guards. For example, it is very helpful to specify preconditions and post-conditions for methods. Or to note down boundary conditions for fields (e. g. never be < 0). These boundary conditions are then also important for the automatic further processing of UML, for example.
- UML 2.0 Diagram Interchange: A fourth part specifies the layout of the diagrams. With this specification, the layout should no longer be lost with the exchange of diagrams across tools, since each diagram is described by means of a graph (node, edges, sheets).

1.3.2 How to use UML?

UML may be a nice notation, but many people will not be able to see which areas of application there are at all.

Communication and Abstraction

The real purpose of UML is to communicate ideas! The question is always, how can you best and fastest bring the design of a system across and document it. It is clear that UML may not always be the best tool and may not be complete.

Therefore, there are many critics who say that UML is too big and too complex. UML tries to please everyone. Nevertheless, UML has established itself against all its previous notations. It is the language that has become established in the industry and which you should definitely master, but still take a critical look at.

Another important benefit of UML is that it allows you to hide details and refine them later with or without UML. In many cases, you have to approach the system Top-Down and specify specific parts or components from the vision / analysis.

How do they succeed? Here UML diagrams can help to define the rough structure/ requirements and then to define them in finer diagrams. Often, however, you simply have to talk to other designers, analysts, stakeholders, project managers or clients who sometimes have no idea about IT. How do you present the system to them? Although many of the people in the community do not have comprehensive IT knowledge, most of them are familiar with UML and the meanings of some diagrams. You will soon be able to explain the meaning of a use case or deployment diagram well enough to convey the basic structure of your system.

Reverse Engineering

One of the possibilities for using UML is Reverse-Engineering. You may already have a system that needs to be further developed and you want to get an overview of the system. Good tools can analyze the existing code and generate various UML diagrams from it. It can be very helpful to get out of a mountain of code, first of all, some diagrams (like. B. class, package or component diagrams) and thus to understand the program' meta'.

Reverse engineering is a not to be underestimated approach in the industry when it comes to achieving competitive advantages. A well-known example of reverse engineering is the Lotus 1-2-3 spreadsheet, the program has been' decomposed',' understood' by Microsoft and rebuilt as MS Excel. This is also reverse engineering.

Sketch- / Blueprint-Mode

In addition to the written documentation of an architecture by architects, UML is often used more informally as a sketch in team discussions. In sketch mode, informal UML is used and the UML diagrams shown mostly show only the interesting sections of the system that is being discussed.

In the first mentioned Blueprint mode, the architect of the system is usually busy with more or less completely depositing the design of the software (blueprint = blueprint; a term from the architecture).

Sketch and blueprint mode are not synonymous to use. Sketch is a small discussion about clippings. Blueprint is a larger architectural specification.

Programming language / MDA

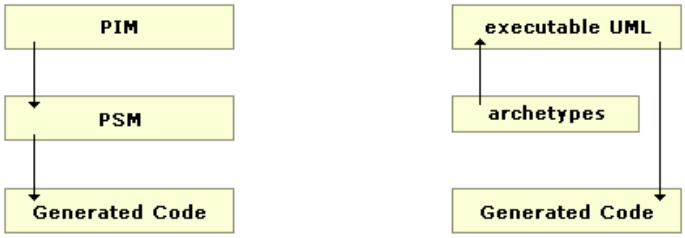
An important element of UML is that it can be processed by machines. The UML and the associated languages such as XMI (XML Metadata Interchange) are read by Frameworks, which can also generate code.

The development of software is relatively expensive, especially European IT companies need to see how you can be competitive and fast, qualitative (!) high quality software.

This can be done by automating the generation of code. If UML diagrams describe the components, packages, classes and also the behavior of software, why not generate as much code as possible? In this case, the UML itself would be something like a higher quality meta programming language.

This approach leads us to the research area of the MDA (Model Driven Architecture), for which there are already an astonishing number of powerful frameworks. Unfortunately, many of these toolkits are hidden in the industry. There are also some good free toolkits available. B. under AndroMDA (http://www.andromda.org/docs/index.html).

MDA is part of the module "Model-Based Software Design" and is treated more intensively there.



Model Driven Architecture (MDA)

The most important branch is shown on the left. You model in UML. You can create a model - the Platform Independent Model (PIM) - which is more concrete, but still independent of the underlying technology.

In the next step, you would generate a PSM (Platform Specific Model) (all with MDA tools) that contains concrete and executable code for a platform. Z. B. Code for a. NET or J2EE platform. This means that there are indeed frameworks that can generate executable code. Sometimes even executable applications that work a lot with templates or prefabricated building blocks.

Another approach is executable UML. The intermediate step of the PSM to be created is skipped and the code is generated immediately afterwards. Archetypes help to describe how the encoding step from UML to code should be done.

1.3.3 Design Flow

When dealing with design, the initial difficulty is often knowing what the concrete procedure looks like. It is important to clarify which diagram should be selected first and why.

Which order is best?

Of course, this question is difficult to answer and depends 100% on the system to be designed. Nevertheless, here is a suggestion based on project experience and considering the abstraction level of the diagrams:

Beispiel

Abstraktionsgrad von Diagrammen

1. Use Case D
   The use cases are certainly among the most abstract. What the user wants from the system should usually be known first and should be part of the analysis rather than the design.

1. Deployment D
   As here is shown, where which coarse component is running, this is also very abstract. This is about physical locations for large projects. So server X here, many clients there, a billing server there, etc. This is abstract and usually belongs far forward.

1. Package D
   Packages usually contain components that consist mainly of a collection of classes. Therefore, component diagrams must be created before class diagrams.

1. Class D
   For this reason, class diagrams should be created according to packages and component diagrams.

1. Composite-Structure D, Subsystem D, Object D
   The diagrams mentioned here are not so often used and are often scattered throughout the entire design phase.

1. Activity D
   The activities of an application are usually specified much earlier in this order to clarify the flow structure of the application. This can often be the prerequisite for the designer to get the components clear. Sometimes even before the deployment diagram but after the use cases. For the sake of clarity, it is located here according to the static diagrams in the dynamic diagrams.

1. Sequence D
   They show the behavior of components or classes. Generally, components or classes must exist before the sequence diagram can be modeled.

1. State D
   The state of the application, components or classes can also be modeled at any time during the design phase. Usually, after the use cases, deployment and activity diagrams have been created."

1. [Interaction-Overview D, Timing D]
   These not so often used diagrams can also be used everywhere if required.

So it would be ideal now if you take your own motivating (private) project and model it and accompany it over a longer period of time. Perhaps the above-mentioned orientation in the design process and the explanations of the concrete diagrams in Chapters 1.4 and 1.6 may be helpful.

1.4 UML Evaluation

The meaning of UML is much more controversial in reality than it seems.

In the academic field, UML is taught in many places, as there are still many proponents on the university side. It is not uncommon for industry to accept this uncritically.

Nevertheless, there are companies and movements that rate the usefulness of UML much less. Especially in the agile environment, other techniques are therefore often used. This can be seen in the meanwhile quite "old" XP model, which does not include extensive UML artifacts. In some major companies, such as e. g. Google is even banned UML!

This does not mean that UML is completely prohibited. If two developers want to "talk" to UML, this is no problem. However, the power of a prototype or prototypical architecture in corporate culture is seen as much more efficient than a UML paper.

It is therefore crucial for every software engineer to know the spectrum of tools or communication tools and to be able to judge for himself to what extent UML is a suitable tool or other methods are better suited.

1.5 Statische Strukturdiagramme

Title: Static Diagrams

This section presents the static structure diagrams:

- **Deployment Diagram**: How are the software components physically distributed?
- **Component Diagram**: What units / components does the software system consist of?

- **Package Diagram**: Which concrete packages / namespaces are used to hierarchize the components?
- **Class Diagram**: Which classes are there (in a namespace)?
- **Composition Structure Diagram**: How are components related?
- **Object Diagram**: Which objects exist at a certain point in time?

1.5.1 Verteilungsdiagramm

Title: Deployment Diagram

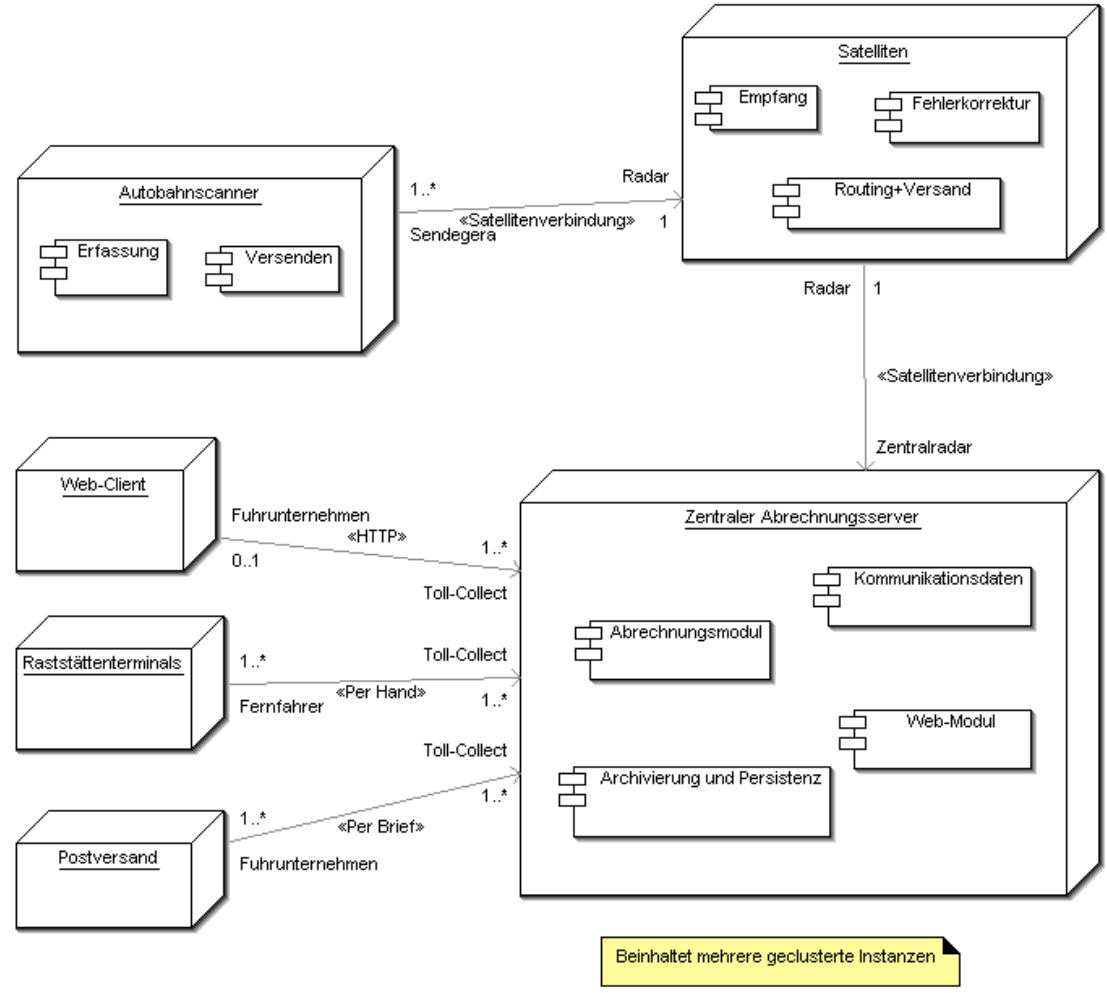
The distribution diagram Deployment Diagram) is specified relatively early. Relatively soon after the Use Cases. This diagram is about showing the physical structure of the entire application. This means which components are running on which physical devices and where.

This is often difficult for students to understand, because you usually only write small projects that run on one computer.

For example, the TollCollect project includes various data centers, satellites, equipment on motorways and equipment in motorway service stations. So it runs very different software on very different computers in many places.

The aim of the distribution diagram is to present the physical distribution conclusively.

Beispiel



Verteilungsdiagramm des Systems Toll-Collec

We see here an example of the German toll system Toll-Collect. Trucks are registered at the motorways. This data is then transmitted to the control center by satellite. The components in the central system show which tasks the components perform. For example, users can either go to the service stations for payment or pay by web or by mail. This example is intended to help you understand the physical distribution of such large systems as the toll system.

Notation

The individual physical locations/nodes (if necessary Servers) are represented by boxes/cuboids. The boxes are connected by simple lines and represent the communication between them. Components or artifacts can be displayed in the boxes. For example, artifacts can be files of different formats such as: jar, dll, war.

The connections of the nodes describe the type of communication connection. You can also specify multiplicities. For example, you can specify that up to 1000 clients can connect to the server at the same time.


Hinweis

Hint

1. For practice, it is advisable to simply model a larger system than what you later want to build or specify more precisely. This results in a more extensive distribution diagram. Or plan your application as a more complex distributed system.
1. Imagine the system administrator comes to you - he knows only that there is work coming to him - and wants to know which hardware he has to provide. The distribution diagram helps him to do this. With the additional components in it, he also has the possibility to understand what the system should do roughly and can also estimate the costs.

1.5.2 Komponentendiagramm

Title: Component Diagram

The Component diagram is specified relatively early. Relatively soon after the use cases. This diagram is about showing the physical structure of the entire application. This means which components are running on which physical devices and where.

This is often difficult for students to understand, because you usually only write small projects that run on one computer.

For example, the TollCollect project includes various data centers, satellites, equipment on motorways and equipment in motorway service stations. So it runs very different software on very different computers in many places.

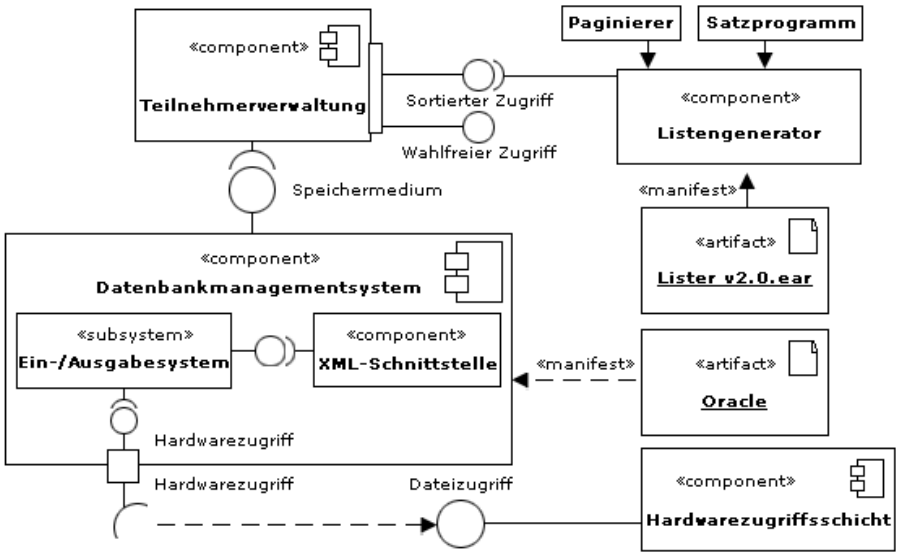
The aim of the distribution diagram is to present the physical distribution conclusively.

Bernd Österreich
writes: *"A component can be instantiated similar to a class and encapsulates complex behavior. It is used to form units with a high degree of professional coherence. In contrast to a class, the principle of interchangeability (substitutes) is also aimed at for a component. A technical platform for components, for example, is Enterprise Java Beans (EJB)."*
[Oes98]

Notation

The component is a rectangle with the typical' Lego symbol' in the upper right corner. It contains the keyword
<<component>>
, that is often not shown for reasons of clarity.

It is allowed to insert other elements - such as objects or components - into the object, but this can easily lead to very complex diagrams (packable element).



Beispiel Komponentendiagramm

What does this diagram show?

- Some Components Komponenten (with the Lego-Symbol)
- On the right some artifacts. This can be paper or software.
- Interesting are the balls, into which the semicircle reaches. The ball is the provided interface. The semicircle is the required interface.
- Showing small squares there are still 'ports' that allow access to the component, which is not based on interfaces.

In diagrams, the keywords from different UML variations still haunt the keywords
<<realize>>
or
<<reside>>
(from which class this component is implemented) or
<<implements>>

. The latter describes the concrete interface that this component also implements.


Anmerkung

Again, one can imagine that this diagram is not only used for internal communication, but also for dialogue with developers and clients.

A few examples:

- The client wants to know how the new components interact with the old ones.
- The customer wants to know and then decide for himself which components (or modules) he can buy when and how they fit together.
- The junior developer wants to know in which components the z. B. Dismantle the full-text indexing system. He asks how that works, how to build it. You draw a diagram with components such as parsers, configuration of the search engine, configuration of the external modules, activation of the engine, handling and preparation of the results, etc. Result: The junior developer understands and has a vision of the structure.

Often Design Patterns as MVC are reflected here.


Wichtig

In this context, it is important to note that in UML there is also the stereotype Subsystemwhich is often used in component diagrams. Here, only a container for other components is built up - represented as a simple box with the stereotype Subsystem.

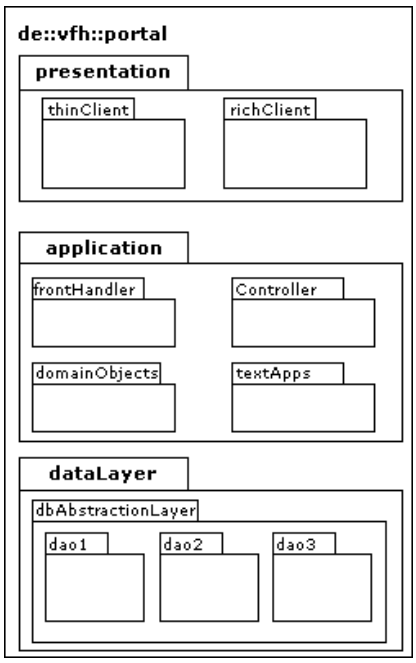
1.5.3 Paketdiagramm

Title: Package Diagram

The package diagram structures the namespace of packages, components, or classes.

It usually corresponds with the packages in Java or with the namespaces in C#. For example, de.vfh.portal.database which includes the class AccessLayer. It is important to have a hierarchical structure that allows the unique identification of the model element.

Beispiel



Paketdiagramm

In the example image you can see that all drawn packages should be placed under the de::vfh::portal namespace.

At the top you can see a presentation package. It contains a package for a rich client, such as a Swing GUI and a thin client, such as a JSP application.

In the middle you can see the application package, which contains four sub-packages. A front-handler for communication with the GUIs, a controller for flow control, the domain objects (i. e. the business objects) and a package for management with external applications.

Below you can see different layers for the database connection.

The package diagram shows the hierarchical structure of the model elements, showing layers and component blocks.

In the notation there are still package merges, which are not explained here and are not very important in practice. Try to find something yourself.

Anmerkung

Why are package diagrams so important?

- In practice, package diagrams are important because the developers want to ask for a "playground" early on and develop test code.
- Packages can reflect the real architecture of the system like component diagrams. Design patterns and layers in the application become transparent here. It is important that transparency is provided at an early stage, that there is sufficient discussion about it from the outset or that experienced designers are employed.

Package diagrams are most important because they allow you to model dependencies. So try to draw the package and ask yourself: Who is using whom? Use arrows to draw this into the diagram. This is shown in the' Usage Structure'. This usage structure is extremely important for good design and architecture.

Is there a completely confused structure here, who is using whom? Or is there a clear flow from top to bottom? The latter is very important for a changeable architecture.

For example, in a package diagram that contains 'uses' relationships, you can also see very nicely whether there are cycles in use. So package A uses package B and vice versa? If this is the case, a redesign is often necessary.

In the last chapter, we see that there are many tools that automatically analyze the usage relation between packages and point out problems.

But experience this for yourself by designing a package diagram in such a way that there is a hierarchical usage relationship in addition to the hierarchical namespace. Your code becomes more maintainable and insensitive to change.

1.5.4 Klassendiagramm

Title: Class Diagram

A class diagram visualizes classes in different ways. For this purpose, the classes themselves are displayed once, if necessary their relations with each other and possibly with each other. also their internal structure. Usually there are two goals:

Representation of classes and relationships: Here different classes of a package or component are displayed in the diagram. It is important to show which classes exist at all and which classes are used by others. The latter is called associations.

Representation of the inner structure: The author of the class diagram then wants to show or define which attributes or methods the classes have.

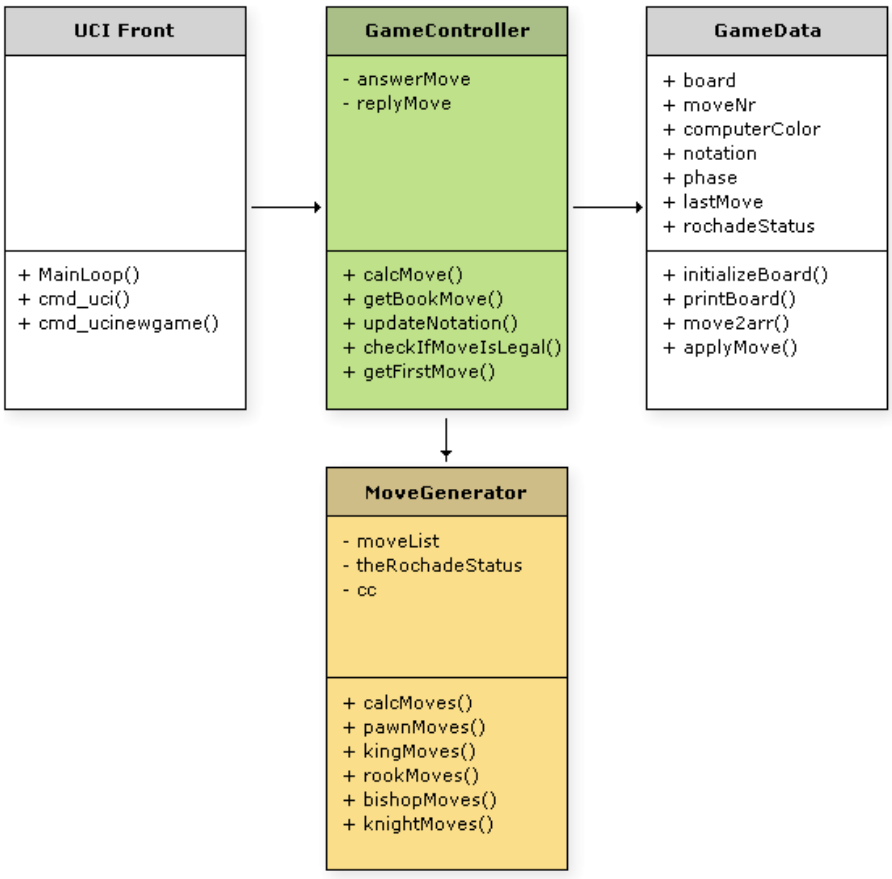
Both forms are often combined with each other and then form a detailed class diagram. But there's more information in the class diagram:

1. What stereotype does the class correspond to? For example, a display class, a controller class, and so on. This will be explained later.

1. Of which structure are the attributes and methods? Which signatures do the methods have? So this is about visibility, like public / private, the types, like int or string or the parameters, like int b, string c.

Beispiel

The following example shows a simple class diagram showing attributes and methods with their visibility (+ and -) but no more extensive associations.



Example Class Diagram

In UML 1, there was a strict distinction between an operation and a method - as the implementation of an operation. However, this is no longer the case since UML 2.

1.5.4.1 Notation

Title: Notation

The class itself is represented in a rectangle containing 1-3 areas. The class name is at the top, which is usually represented in bold type. Below this, the attributes first and then the methods. Class names are capitalized in almost all languages.

Hinweis

It makes sense to specify the responsibility of classes as a comment, i. e. what the class itself is supposed to do, unless this is shown trivially in the class itself.

The following are the attributes that can be specified with or without parameters or visibility. Visibility, types, parameters, constraints or initial values are also optional.

Class attributes are displayed in the same way and belong to each class instance once. This means that all instances can access this attribute.

A Class in the Class Diagram

Abstract classes can be marked in italics or bold. Most of the time
{abstract}
will be written under the class name.

As **Visibilities** we do have

- public **+**
- protected **#**
- private **-**
- und package **~**

These are noted with the symbols above.

Derived attributes can be defined in the UML with a forward slash "/".

It is interesting to note that - unlike in most programming languages - the name is displayed first and then the type (as in Scala or Nim). Don't let this confuse you. An example:
person: String = "Katharina"

This also applies to the parameters of an operation / method! So, that
getMwst (betrag: Integer)

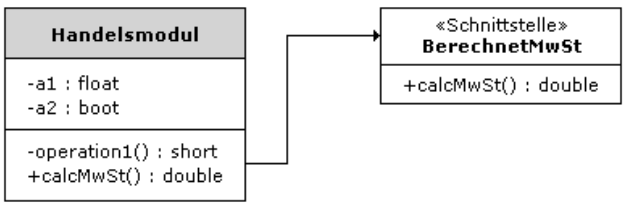
is a valid notation for a method.

1.5.4.2 Interfaces

Title: Interfaces

Yu can mark classes as Enumerations or Interfaces. Enumerations will be denoted as <<enumeration>>. Enumeration quantities (e. g. red, green, blue, etc.) can then be found in the body of the enumeration.

More important we have
Interfaces
, that specify a contract to be implemented in the form of method specifications. They are specified with the keyword
<<interface>>
above the class name.



Interface Example

It is said that the trading module realizes or implements the interface that calculates VAT.

Attributes are not written down in interfaces (but it is not forbidden) because it is about an access contract and not about the inner structure of the class.

Since interfaces are strict contracts, you will find more detailed specifications such as exceptions, invariants, preconditions and post-conditions.

Another possibility to write down interfaces is the representation as sphere and semicircle. The sphere implements the interface and the semicircle is the interface to which it is "docked".

Interface as a Circle and a Semicircle

In addition, you can describe the semicircle with one word, which property this class implements (e. g. B. serializable, or sortable).

1.5.4.3 Stereotypen

Title: Stereotype

Definition

Stereotype
A stereotype is a consistent pattern that occurs or can be reused. The term actually originates from psychology and sociology and refers to the same behaviour patterns of people or groups.

Translated to the UML and classes, this means that you can mark/classify a class or other UML elements with a stereotype. They enable the use of platform or domain-specific notation, which in a way marks them out to implement a certain behavior. For example, for
<<Stateless>>
for a stateless session bean.

Since UML 2, classes can contain several stereotypes. Most UML tools allow you to assign stereotypes directly to elements such as classes, dependencies, components or packages. Stereotypes are often displayed in the same colors.

The UML already specifies a number of so-called "standard stereotypes" that are available for your modeling. The following are some of the relevant standard stereotypes:

span class="courier"><<focus>> states that the classes identified in this way define the central logic or control flow.

- <<auxiliary>> implement secondary logic or secondary control flow and thus support other classes that are central to the model.
- <<<utility>> define a collection of static attributes and class operations that can be accessed from all classes.
- <<create>> can be one of two predefined stereotypes, of which

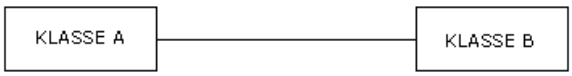
one can be applied to dependency relationships and one to operations.

- <<destroy>> is inverted to create and says that the operation destroys a form of the classifier in which it is contained.
- <<call>> indicates that a dependency relationship marked in this way has operations or classes with operations at its ends, to which the association refers.
- <<instantiate>> means that operations of the consumer generate characteristics of the provider.
- <<interface>> is an interface. It cannot generate any values, it does not contain any attributes, and its operations cannot be implemented in methods.
- <<send>> is applied to dependency relationships and says that the source of the relationship (an operation) sends a signal.
- <<type>> is attached to a class and says that this class groups a set of objects with respect to the operations that can be applied to these objects.
- <<responsibility>> implies an obligation or a contract between two model elements.

1.5.4.4 Assoziation

Title: Associations

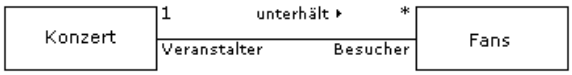
The association describes a relationship between classes in which one object uses the other.



Assoziation

Here is an example. Let's say a customer is the class. Several invoices can be assigned to this class. The Customer object must be able to access invoice objects. Special cases of association are aggregation and composition.

The simple association with a simple line is shown.



Qualified Association

The multiplicities with which you specify the number of links between the two objects are important. Examples are:
0... 1, 0... 42, 1... 1, 1... 42, 0... *, 1... *

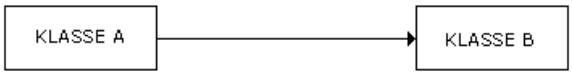
E/R diagrams are often referred to as
1:1, 1: N
or
N: M
- relationships. In general, relations are therefore important hints when specifying objects, database schemas, or database mapping information, for example.

Furthermore, associations can be qualified by marking the line with roles and properties.

And finally, there are multipartite associations in which several classes are linked. In this case, the lines converge into a small diamond.

Directed Assoziation

The directed association describes that you can get to the target class via the source class. But not the other way around. It is therefore a unidirectional relationship.

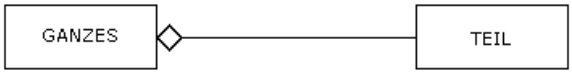


Directed Association

If you want to be able to navigate in both directions - i. e. bidirectional - you can combine two arrows into one. These can then also be provided with roles and properties.

Aggregation

An aggregation is a special case of an association. It is called a part-entire relationship.



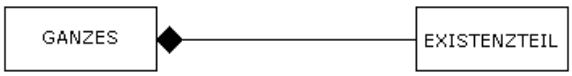
Aggregation

Therefore, you often write "consists of" as a property on the arrow. An example would be a "party" as a whole, which consists of many participants. If the "party" and thus the aggregation is dissolved, the participants are still able to survive. You don't have to have the party. However, the party only results from the aggregation of the party participants.

If you are in doubt as to whether aggregation exists, you can use an association without making a big mistake. However, a clear award is always more readable and useful.

Conposition

In contrast to aggregation, a composition is drawn with a filled rhomb.



Composition

It is the strongest form of aggregation. The parts depend on the existence of the whole thing. The best-known example is the invoice with its invoice items. If the invoice is deleted, the invoice items cease to exist.

Differently with an association (aggregation) to its members. If an association is deleted from the official register, the existing members would like to and can still live on.

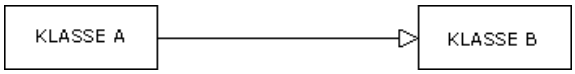
Abhängigkeit

One class requires another. Only then can it fulfill its task.

1.5.4.5 Vererbung / Generalisierung / Spezialisierung

Title: Inheritance / Generalisation / Specialisation

In inheritance, all attributes and methods are also available in the subclass. This is noted in UML with an arrow that is not filled in.



Inheritance

Linguistically, a distinction is made between generalization and specialization.

Generalization refers to the more general properties of the superclasses.

Specialization refers to the subclasses that add further special properties (attributes) and methods to those of the superclass.

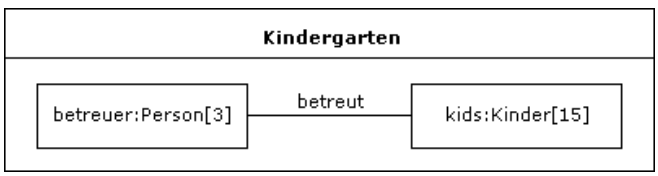
In many cases, the delegation makes more sense than inheritance, as it does not violate the principle of secrecy. In many cases - such as e. g. For example, with graphics libraries - inheritance is useful. Remember to always analyse this yourself.

Multiple inheritance is allowed in some programming languages. In this case, arrows can be combined - or if several subclasses are derived from a superclass (upper class).

1.5.5 Kompositionsstrukturdiagramm

Title: Composition Structure Diagram

Often the available diagrams are not sufficient to show which elements make up a class or component.



Composition Structure Diagram

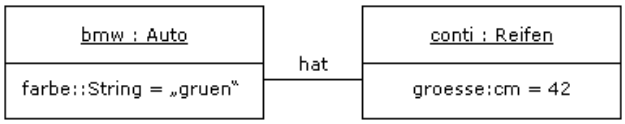
This diagram is drawn with the parent component in a box. This contains two sub-elements. In these elements, the name of the part (part) can be displayed, followed by a colon and a type of the part.

The **multiplicities** are given either in square brackets or superscripted (like the square). The parts can, in turn, be connected to a role.

1.5.6 Objektdiagramm

Title: Object Diagram

The object diagram shows the exemplary state of one or more objects in a system. It is like a photograph of the system in which the objects may have particularly interesting values. It is an instance diagram in which concrete attribute values can be specified.



Object Diagram

This picture shows classes in a certain state.

Just a few more points.

When defining the object class or attribute type, you are very free. These elements can also be simply omitted.

This diagram can also be useful if OCL expressions are specified. This would result in e.g.:
contect tires inv: size > 20
(therefore too small tires are not allowed).

As shown above, links can also be drawn between objects. These are then the associations and are only refined with the role.

1.6 Dynamische Diagramme

Title: Dynamic Diagrams

Now we continue with dynamic diagrams:

- **Use-Case Diagram** → What are the Use Cases?

- **Activity Diagram** → What is a way through the application?
- **State Diagram** → What states can the program be in?
- **Sequence Diagram** → How do classes / objects communicate?
- **Interaction Diagram** → An Activity Diagram which replaces some activities with interactions.
- **Communication Diagram** → Represents complex communication processes
- **Time Diagram** → State lines for theEmbedded- or the electronic

field.

1.6.1 Anwendungsfalldiagramm

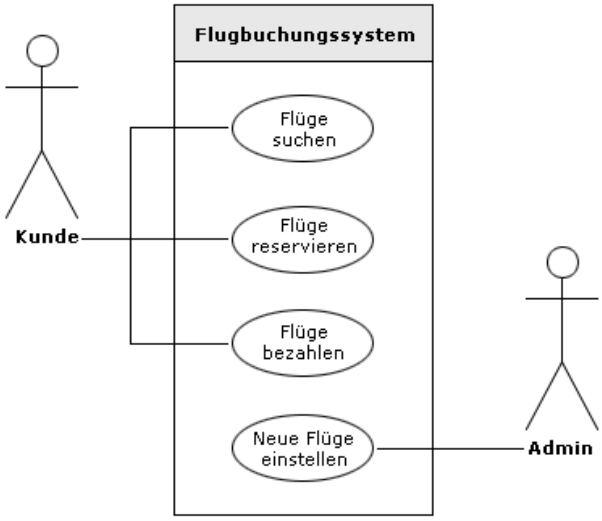
Title: Use Case Diagram

This diagram is quite important and is almost the first UML diagram to be drawn in projects.

The goal is to show what you want to do with the software system to be built. Which cases of application there are therefore! From the analysis phase, it is known that application cases need to be documented. This describes the workflows in the system that should be possible. In the UML, this is now to be supported even more easily with simple graphics.

The diagram shows so-called actors as stick figures. These can be people such as customers, administrators or any other users.

From there, strokes go to the respective use cases, which are written into an ellipse. To group them together, one notes related use cases in a box. Let me give you a very simple example:



Anwendungsfalldiagramm

The use case diagram is intended to communicate the use cases as simply as possible - for example, to management. Therefore, it is usually sufficient to look at such a diagram, and the viewer can immediately develop a vision of the system himself or suggest his own ideas. This is often more efficient than reading extensive application case documentation.

The strokes between the actors and use cases can be named as in the class diagram and provided with cardinalities. For example, only two pilots at a time want to have programmed the "Autopilot switch off" application.

Note in the above example that you have the feeling that the actions take place from top to bottom. But application case diagrams are not directed! Activity diagrams are available for the chronological sequence. Mixing the diagram types is a typical bug of UML beginners. Connecting complex use cases in a Use-Case diagram with many arrows is not correct.

Weitere Anwendungsfallbeziehungen

1. A dashed arrow indicates that there is a realization. This means, for example, that the use case "Go to reporting point" can be realized by two cases connected with a dashed arrow, which can be called "drive to reporting point by car" and "walk to reporting point". Together, the two of them make up the main application case.

1. If you write

   <<include>>

at the dashed arrow , the case of application to which it is pointed is included. It's like programming languages do. Thus, if the application consists of three internal parts A, B, C, then part A can be outsourced as an application and replaced with

<<include>>

the dashed arrow to show the important include again.

1. With

   <<extends>>

   and a dashed arrow, it is said the other way round that an application can be extended with another one. The arrows always go from the external / include object to the main object that uses it.

1. A specialization arrow (analogous to that in the class diagram) indicates that several "sub-application cases" constitute a more general use case. For example, a use case can be the "standard authentication", which is carried out either with an "electronic ID card authentication" or with a "mobile phone number authentication".

1.6.2 Aktivitätsdiagramm

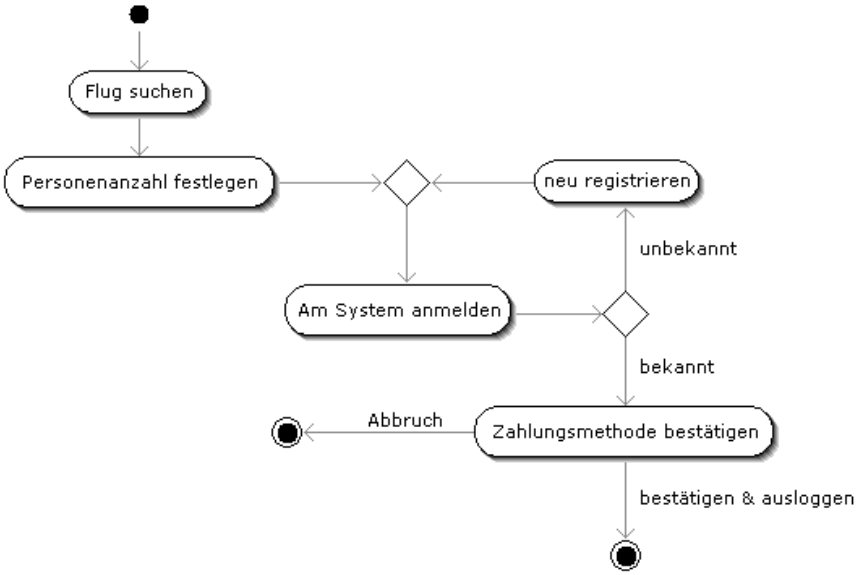
Title: Activity Diagram

The Activity Diagram is one of the most complex diagrams with its many elements. Not all elements can be played here, but they are easy to look up in the UML overview of OOSE. We are confining ourselves to the most important elements.

uml-2-Notationsuebersicht-oose.de.pdf
(304 KB)

The activity diagram is usually used early in the modeling phase and shows the sequence in which the user is guided through a possible path of the application. There are action nodes, object nodes and control nodes. The latter allow you to branch out and make decisions.

Activity diagrams visualize and explain the program's flow, which is often very difficult to visualize with the natural language or the written word.



Aktivitätsdiagramm

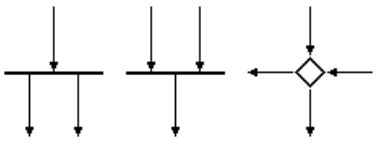
In this diagram you can see some action nodes (called activities at times of UML 1). Furthermore, a start node and an end node. On the right there is a branching. Object nodes are not drawn. In this way, you could insert objects in various places, which are then available at this point.

For example, a flying object that is selected by the customer after a successful search. Or an object representing the entire flight including customer and payment data.

As in many UML 2 diagrams, it is possible to place diagrams in a box and thus illustrate that an activity can consist of several substeps.

Kontrollknoten

Branches, merges or decisions are represented by control nodes. For merges, conditions can be specified and for branches, conditions can be written to the outgoing arms.



Kontrollknoten

For merging nodes, conditions can be formulated like this:

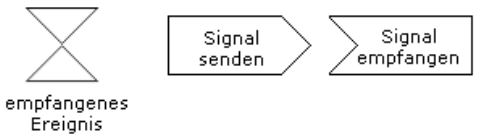
{joinSpec=(X or Y) and (M or N)}

.

For branching, conditions can be specified in square brackets: [X<0].

Weiterführendes

1. **Parameter Pins Object Knots**: Pins represent In and Out parameters for objects.
2. **Swimlanes**: can represent areas, as can be the case in sequence diagrams.
3. **Signale**: Signals can be embedded in the control flow of many diagrams. These are shown as shown below.



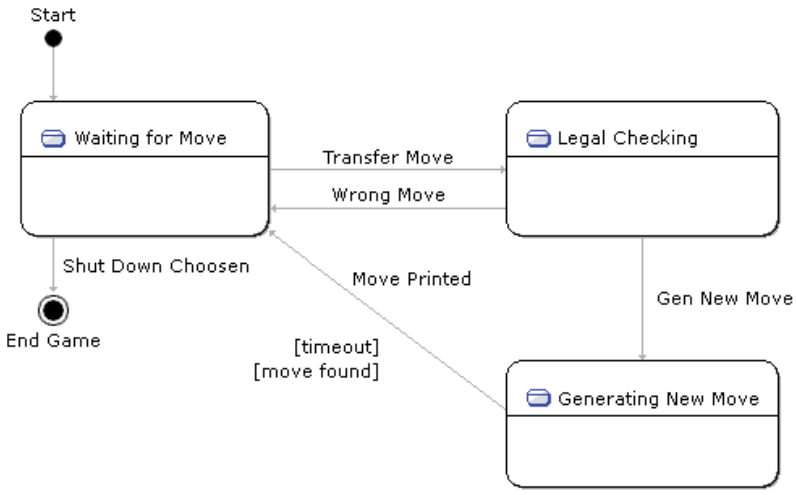
Signale

1.6.3 Zustandsdiagramm

The state diagram describes the behavior of a system in response to certain events. Such a state machine can be traced back to old modeling attempts of Mealy and Moore machines.

It contains:

- States (including compound states and whole substates)
- Transitions
- Regions / Areas
- Start and end state



State Diagram

The notation possibilities differ greatly from application to application. In the simplest case, there is only one box for each state in which the state is described. In this case, the horizontal line should be removed (unlike in the picture here). The three simplest states of a (chess) program are shown here, which can have three states.

Each diagram is usually framed by a box in which the name of the whole diagram is shown at the top.

Furthermore, it is possible to refine use cases in the use case diagram by using state machines. This is often done during authentication, for example.

Notationselemente

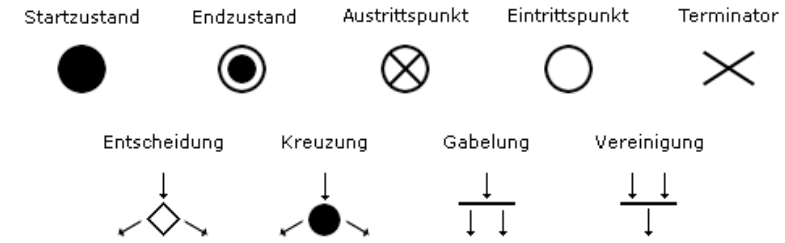
A state is usually (but optionally) given a comprehensible name in the upper box.

The following information can be given in this state:

1. **entry** / Activity
2. **exit** / Activity
3. **do** / Activity
4. **Trigger** / defer, we distinguish:
   - SignalTrigger
   - CallTrigger
   - TimeTrigger
   - ChangeTrigger and
   - AnyTrigger for all

A **Guard** is a condition that can be specified at a transition.

Other elements are already known from other diagrams (such as activity).



Diagrammelemente

1.6.4 Sequenzdiagramm

Title: Sequence Diagram

Sequence diagrams describe a dynamic view of the system. You use it to show how classes interact.

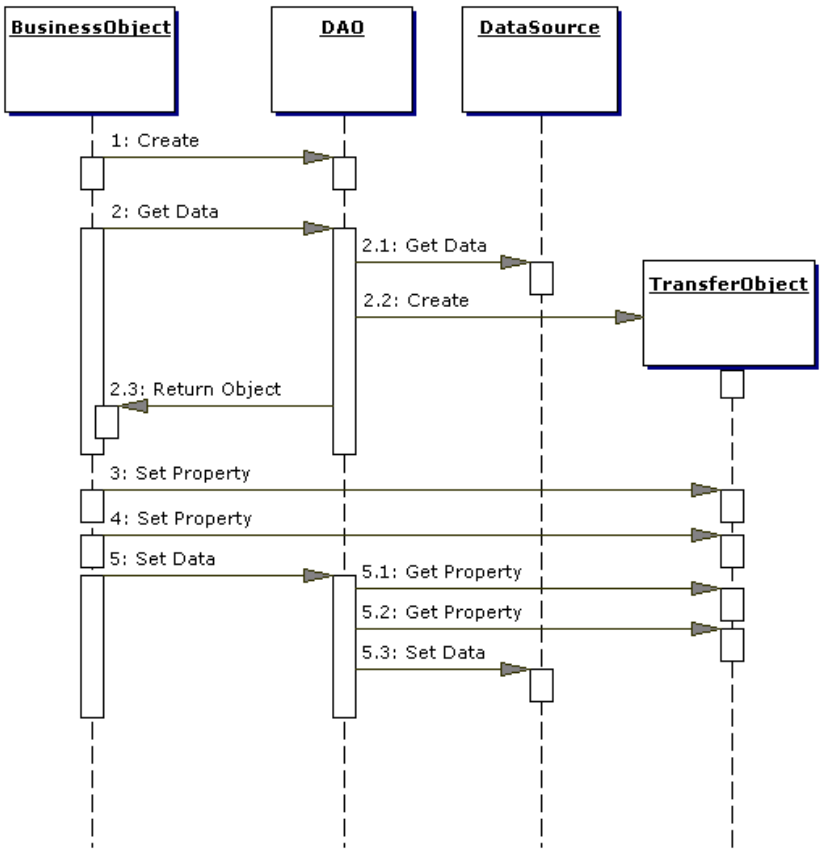
Since the method calls between classes are also referred to as messages that are sent, the diagram displays the messages in chronological order. At the top of the screen (underlined) objects are shown, but classes or even coarser units are also possible. They have lines pointing downwards (so-called swimlanes) or lifelines.

On these lines you can see in blocks how objects are instantiated. Messages from them then go to other objects (blocks). These are displayed as vertical arrows. Messages can have return arrows. The diagrams allow the display of synchronous (filled arrows) and asynchronous messages (non-filled arrows).

At the same time, signals can be received as in the previous diagrams, or branching, iterations and recursions can be displayed. For this purpose, boxes are drawn into the diagram and then, for example, loop or alt (if) are drawn.

An initial found message can start the sequence diagram.

The following diagram of SUN Microsystems shows one of the best and most typical applications for sequence diagrams - Design Patterns. A good architecture for data access is shown here.



Sequence Diagram

What's happening here? The DAO pattern is shown. It contains four objects. Let's take a quick look at it. Not to understand the pattern, but to show why sequence diagrams are useful.

1. A business object requires data from the persistence layer (the data access layer, i. e. ultimately from the database).
2. It is addressed to a DAO object responsible for all database connections.
3. This DAO knows which data access is currently active and gets the database connection via the DataSource.
4. A data transfer object then contains the data in a suitable form (e. g. better than ResultSets) and this is returned to the business object in 2.3.

What follows from 3 to 5.3 is the reversal, which doesn't have to be of particular interest to us in terms of content. For the sake of form: Instead of writing back data via JDBC, for example, it is now state-of-the-art to change the transfer object and then save it via the DAO.

You have seen the scenario to see how helpful sequence diagrams are. The communication of four important objects in the data retention layer becomes more transparent through the display. Together with the Active Record Design Pattern, the example shows the most important design patterns of the data retention layer.

It's interesting to note that sequence diagrams often have two types of communication scheme:

1. **Centralized Control:** One object does the work (calculates or holds the threads like a spider in hand) and all the others close.
2. **Distributed Control:** All classes / objects are equal and "calculate" as it were. There's no central control.

In many cases, the second structure is more advantageous and allows more object-oriented concepts (polymorphism, overwriting, etc.).

For more information, see the following tutorial:

Tutorial: Sequenzdiagramme http://www.tracemodeler.com

1.6.5 Interaktionsdiagramm

Title: Interaction Diagram

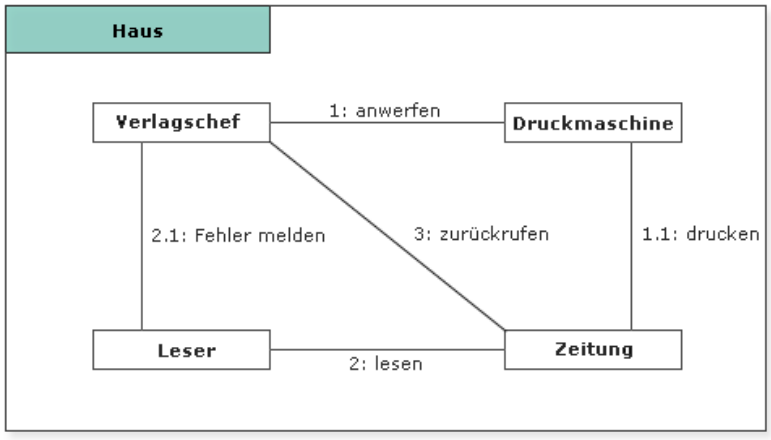
An interaction diagram is a hybrid form of an activity diagram in which individual activities are explained in more detail by an interaction diagram.

In concrete terms, an activity diagram is therefore available in which individual states are to be explained in more detail. These states are then magnified in the diagram and an interaction diagram is embedded, i. e. displayed. This is used whenever states in activity diagrams are very complex.

1.6.6 Kommunikationsdiagramm

Title: Communication Diagram

The communication diagram represents more complex communication processes.



Communication Diagram

What we see here is the attempt to depict a complex sequence in the correct order.

The first communication line is the newspaper's start-up and printing by the head of the publishing house. The buyer reads the newspaper and reports a big mistake. The head of the publishing house pulls the newspaper out of circulation.

In computer science, such a diagram is used whenever the contents of states, transitions and data are not important but the elements should be represented in a diagram. This means, for example, that the tasks or data of the press are not important, but only that they exist and how they interact.

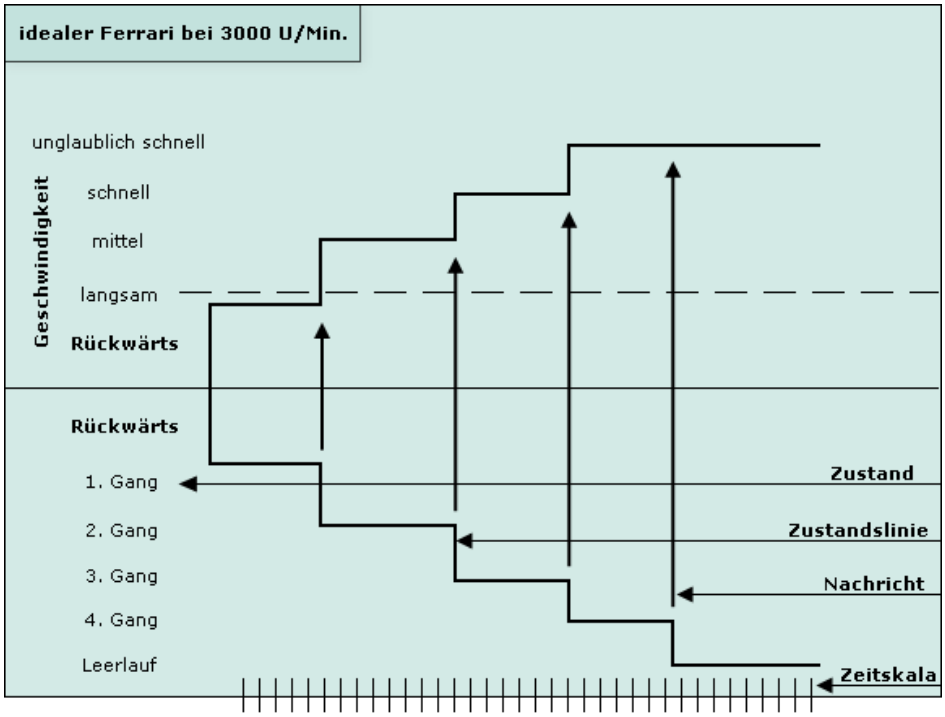
1.6.7 Zeitdiagramm

Title: Timing Diagram

Time diagrams are mostly used for embedded applications or electrotechnical applications. Only in these areas are there such precise processes or procedures that they have to be represented in time. In the otherwise customary modeling of software engineering processes - such as an activity diagram - the chronological sequence does not matter. A booking process can take seconds or even hours.

The diagrams look like signals on an oscilloscope and contain:

1. Time lines
2. Terms and conditions
3. Messages that can lead to a change of state.



Timing Diagram

1.7 Zusammenfassung UML

Title: UML Summary

- As a set of interaction elements, UML serves as a method for developing models for software systems.
- UML helps determine the design process.
- The fathers of UML are Grady Booch, James Rumbaugh and Ivar Jacobsen. They are also called "The Three Amigos".
- UML is not a dogma and is not absolutely necessary for the development of a software system.
- The purpose of UML is to communicate ideas. By using various diagrams, it is possible to illustrate situations that would be difficult to present in writing or in language
- In the UML, different diagram types are used for display. Here is a subdivision into:
- ◦ **Static Structure Diagrams**

(Zur Darstellung der Struktur von Modulen eines Softwaresystems. Die UML kennt sechs Strukturdiagramme: Klassendiagramm, Kompositionsstrukturdiagramm, Komponentendiagramm, Verteilungsdiagramm, Objektdiagramm und Paketdiagramm.)

- **Dynamic Diagrams**, also behaviour diagrams
  (For displaying function sequences. The UML uses seven behavioral diagrams: Activity diagram, use case diagram (also known as use case diagram), interaction overview diagram, communication diagram, sequence diagram, time history diagram and state diagram.

- There are a number of tools that support modeling with UML. However tools describing the UML on a meta level (not drawing) as plantuml have many advantages.

1.8 Wissensüberprüfung UML

Title: UML excercise

Task UML-01: Specification

Download the UML specification from OMG and read / browse through it a bit.

Time needed: 20 Min

Task UML-02: Tools

Make sure you have your favourite UML tool at hand. Evaluate some of the tools mentioned in the session. Allow some time for this. Often you are not satisfied with the first tool. For example, one does not like the generated code or a diagram is missing. If a tool suits you, practice with it and make sure you feel fit in it!

Time needed: 60 Min

1.9 Literatur UML

Title: UML Books

When designing, working with UML tools and in project work it is extremely important to use good literature to look up the subtleties and meaning of the notation elements correctly. The books listed here also contain many examples.

Three works that were also helpful in the creation of the module are worthy of special mention:

Martin Fowler:

"UML Distilled"

Pearson Publishers

Jeckle, Rupp, Hahn, Zengler, Queins:

"UML 2 glasklar"

Hanser

Bernd Österreich:

"Analyse und Design mit UML 2.1"

Oldenbourg