UML – Notation für objektorientierte Systeme

Die *Unified Modeling Language* (UML) ist eine graphische Notation, eine „visuelle Modellierungssprache“ [UML Referenz], die es erlaubt die „Artefakte“ eines Softwaresystems von den Anforderungen über die Spezifikationen bis hin zum Code

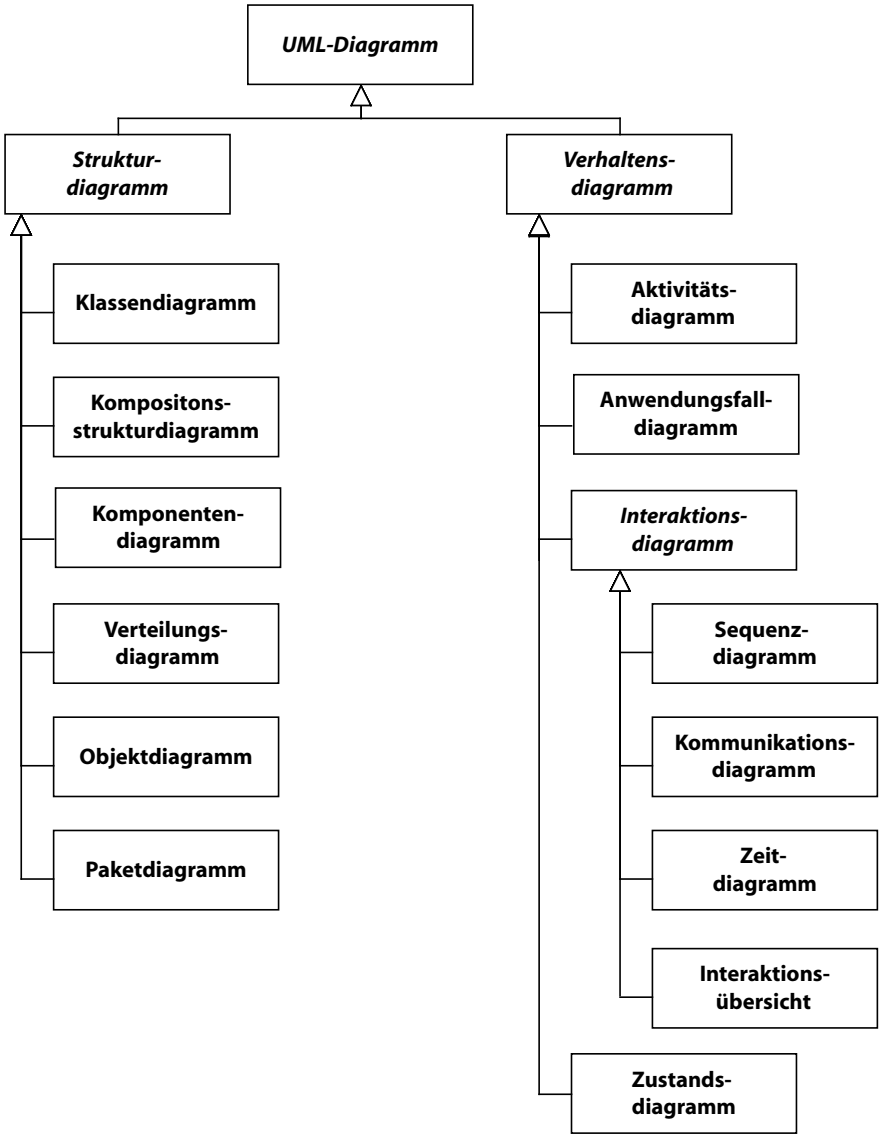
- zu spezifizieren,
- zu visualisieren,
- zu konstruieren,
- zu dokumentieren.

Ursprünge der UML sind die objektorientierte Programmierung und der Entwurf von Informationsmodellen und datenbankschemata (insbesondere die Entity-Relationship-Diagramme), die zu Beginn der 90er Jahre des letzten Jahrhunderts in verschiedene Notationen und Methoden der objektorientierten Analyse und des objektorientierten Designs (OOAD) mündeten. Prominente Vertreter solcher Notationen waren Peter Coad, Grady Booch, James Rumbaugh und Ivar Jacobson. Die drei zuletzt Genannten fanden sich schließlich unter dem Dach einer Firma vereint (Rational Corp. - heute IBM) und verschmolzen ihre Notationen zur UML. Mit geringfügigen Änderungen wurde das Ergebnis der Arbeit der „3 Amigos“ von der Open Management Group (OMG) 1997 als Standard verabschiedet.

Seither wird die UML von der OMG, einem Konsortium mit etwa 800 Mitgliedern (IBM, HP, Microsoft, Daimler ...), weiterentwickelt: Weit verbreitet und durch viele Tools unterstützt ist die Version 1.5. Aktuell ist Version 2.4, siehe Webseite der OMG: `http://www.uml.org`.

Der Kern der Darstellung eines Systems mit der UML (sei es als Bauplan oder zur Dokumentation) ist ein Modell des Systems. Dieses Modell kann man in der UML durch verschiedene Sichten (eben:) sichtbar machen. Diesen Sichten entsprechen bestimmte Diagrammtypen, die jeweils bestimmte Aspekte des Systems (oder des Modells) besonders gut darstellen.

Die UML 2.0 hat eine ganze Reihe von Diagrammtypen, die sich so einteilen lassen:



Man sieht, dass die UML eine recht umfangreiche Notation ist, und es ist kaum möglich, sie vollständig zu überblicken. Sie enthält auch viele Details, die subtil interpretiert werden können und gelegentlich zu Diskussionen unter Entwicklern Anlass geben. Ich möchte in dieser Veranstaltung nur die (im Augenblick) wichtigsten Diagrammtypen ansprechen. Und von ihnen jeweils nur wiederum die grundlegenden Bestandteile. (Das ist keine arge Einschränkung, weil meist auch nicht viel mehr benutzt wird.)

Wir betrachten heute folgende Diagrammtypen:

- Klassendiagramm

- Objektdiagramm
- Sequenzdiagramm
- Zustandsdiagramm

Außerdem werden später in der Veranstaltung noch zwei weitere Diagrammtypen eingeführt:

- Anwendungsfalldiagramm
- Aktivitätendiagramm

Klassendiagramm

Das Klassendiagramm beschreibt die *Struktur* eines Systems. Dabei sind zu unterscheiden:

Codestruktur: Objektorientierte Systeme werden in der Regel mit objektorientierten Sprachen implementiert. Infolgedessen ist der Code in der Form von Klassen organisiert. Das Klassendiagramm stellt somit die Übersicht über die Klassen und ihrer Beziehungen dar, mithin die *Codestruktur*. Aus dieser Sicht enthalten die Klassendiagramme die wesentlichen Informationen, die zum Verständnis der Struktur des Codes benötigt werden. Die UML erlaubt es nur diejenige Informationen anzugeben (und andere auszublenden), die für den gewünschten Detailgrad erforderlich ist.

Zeigt ein Klassendiagramm die Codestruktur, so ist in der Regel eindeutig, was damit gemeint ist, weil viele der Konstrukte der UML-Notation recht direkt in Konstrukte des Codes, also der jeweiligen Programmiersprache umsetzbar sind. Die Semantik der UML-Diagramme wird durch die der verwendeten Programmiersprache gestiftet. Diese Umsetzung der UML-Notation in Code wird auch durch Werkzeuge unterstützt – man kann in der Entwicklung von Code zwischen dem eigentlichen Quelltext und der Modellsicht wechseln. Auf diese Weise dient die UML zugleich als Notation für die Spezifikation des Codes, der Entwicklung des Codes und seiner Dokumentation.

Es gibt aber auch noch eine andere Verwendung des Klassendiagramms:

In der *Analyse* werden Klassendiagramme verwendet, um Konzepte des Anwendungsgebiets, also der wirklichen Welt darzustellen. (Wir erinnern uns an den zweiten Ursprung der UML aus dem Entwerfen von Informationsmodellen.) Aus dieser Sicht stellt eine Klasse also eine Entität, ein Konzept, den Typ von Dingen etc. im Anwendungsgebiet dar. Das Klassendiagramm dient der Beschreibung der Gegebenheiten der

wirklichen Welt (und nicht des Programmcodes.) Oft stehen bei dieser Verwendung der Klassendiagramm die Klassen und ihre Attribute im Vordergrund, wohingegen Methoden und Details wie Sichtbarkeitsbereiche u.ä. eine untergeordnete Rolle spielen.

Es ist in der Literatur umstritten, wie einfach der Übergang von der *Analyse* zur *Codestruktur* im Prozess der Entwicklung eines Softwaresystems ist. Manchmal wird argumentiert, dass zu den Analyseklassen nur weitere Klassen hinzukommen, das Modell der Codestruktur also nur eine Ergänzung das Modells des Anwendungsgebiets sei. Andererseits gibt es Vorgehensweisen, die auf den expliziten Unterschied der beiden Denkweisen Wert legen.

In der Praxis stellt sich heraus, dass es an der Art des jeweiligen Anwendungsgebiets liegt, wie weit die Beschreibung des Anwendungsgebiets von der Codestruktur entfernt ist. Wichtig ist, wie immer sich die Sache im jeweiligen Falle verhält: Softwareentwickler müssen sich des Unterschieds bewusst sein und wissen, worüber sie reden – über die Sache selbst, oder über den Programmcode.

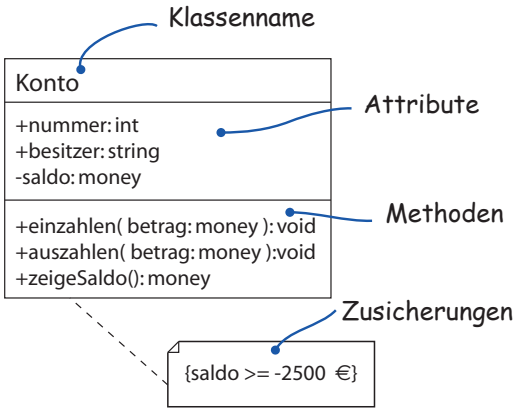
Doch genug der Vorrede, kommen wir zur UML.

Klasse

Die *Klasse* dient der Gruppierung von Attributen und Methoden zu einer Einheit, sie ist eine Schablone, aus der Objekte erzeugt werden können, die einen Zustand mit konkreten Attributwerten haben und Methoden ausführen können, die diesen Zustand eventuell verändern.

In der UML wird eine Klasse durch ein Rechteck mit mindestens einem Abteil dargestellt: es enthält den *Klassennamen* – in der Regel mit einem Großbuchstaben beginnend.

Das Rechteck kann weitere Abteile haben, das zweite für die *Attribute*. Die Attribute sind die Eigenschaften, die Objekte der Klasse besitzen; oder anders ausgedrückt: sie sind die Variablen für die Werte, die den Zustand des Objekts repräsentieren. Die Attribute einer Klasse `Konto` sind etwa die Kontonummer, der Name des Besitzers usw.



Attribute werden in folgender Syntax angegeben:

`<Sichtbarkeitsbereich><Name>:<Typ>`

dabei

- Sichtbarkeitsbereich kann sein `+` für `public`, `#` für `protected` oder `-` für `private`
- Der Name des Attributs beginnt in der Regel mit einem Kleinbuchstaben
- Der Typ des Attributs ist ein gegebener Typ oder ein benutzerdefinierter Typ

Von diesen Angaben können alle bis auf den Namen auch weggelassen werden. (Sie werden vielleicht erst später festgelegt, oder sie werden in der Darstellung ausgeblendet.)

Das dritte Abteil im Rechteck für eine Klassen enthält die Liste der Methoden. Die Methoden werden in folgenden Syntax angegeben:

`<Sichtbarkeitsbereich><Name>(<Parameterliste>):<ReturnTyp>`

und die Parameter:

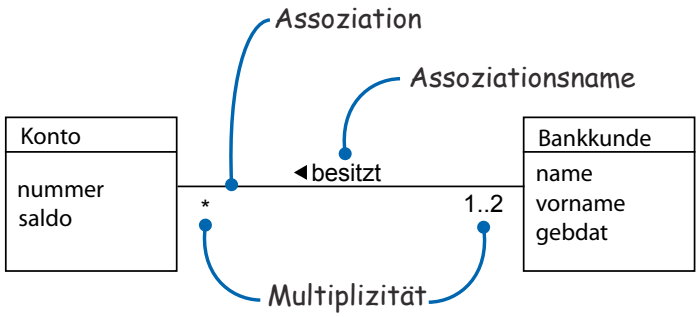
`<Name>:<Typ>, <Name>:<Typ>`

Wie bei den Attributen muss nicht alle Information vollständig angegeben werden, es ist mindestens nur der Name der Methode erforderlich.

Im Beispiel der Klasse `Konto` ist noch ein Kommentar zu sehen, den wir mit jedem beliebigen Element eines UML-Diagramms verbinden können und damit zusätzliche Informationen darstellen können.

Assoziation

Assoziationen setzen Objekte von Klassen in Beziehung. In unserem Besipiel mit dem Konto wird es sicherlich so sein, dass der Bankkunde als Besitzer des Kontos nicht ein Attribut der Klasse `Konto` ist, sondern in einer eigenen Klasse `Kunde` dargestellt wird. Eine Assoziation verbindet nun die beiden Klassen, um auszudrücken, dass ein Bankkunde der Besitzer eines Kontos ist.



Zu einer Assoziation gibt man oft an:

- den *Assoziationsnamen*, eventuell mit Leserichtung
- an den beiden Enden der Assoziation einen *Rollennamen*, der die Rolle bezeichnet, die Objekte der jeweiligen Klasse in der Assoziation spielen – wichtig bei *reflexiven Assoziationen*
- die Multiplizität, die angibt, mit wievielen Objekten jeweils ein Objekt der beteiligten Klassen verbunden sein kann. Oft kommt man mit den Angabe *beliebig viele* = `*`, *mindestens einem* = `1..*`, *genau einem* = `1` oder *optional einem* = `0..1` aus.
- die *Navigationsrichtungen*

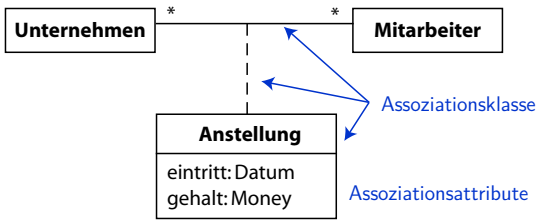
Es gibt zwei besondere Formen der Assoziation:

Die *Aggregation* beschreibt die Enthaltenseins-Beziehung: ein Objekt ist *Teil* eines anderen Objekts. Die Aggregation bedeutet, dass die Beziehung der Objekte *transitiv* und *antisymmetrisch* ist – d.h. es darf keinen Zyklus in der Beziehung der Objekte geben.

Die *Komposition* ist die starke Form der Aggregation, damit meint der UML-Standard, dass die Teilobjekte den Lebenszyklus ihrer Container teilen, insbesondere darf ein Teilobjekt nur zu einem Container gehören, die entsprechende Multiplizität ist also `0..1`.

Entwicklergruppen neigen dazu, sich über den Unterschied zu erhitzen. Das bringt nichts, weil der UML-Standard selbst reichlich schwammig ist. Besser sollte man sich darauf verständigen, nur eine der beiden Formen zu verwenden und ihre Bedeutung genau festzulegen.

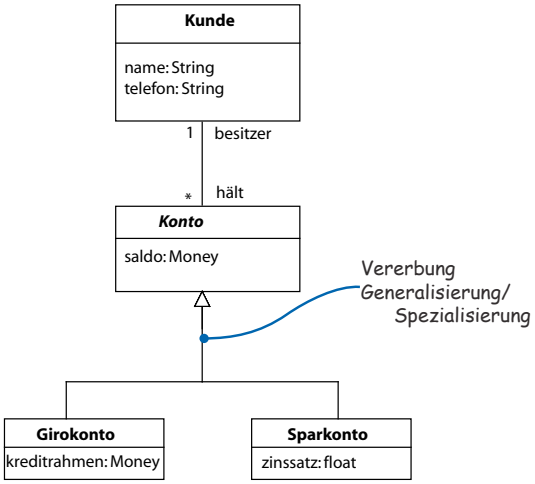
Verwendet man die UML für die Beschreibung von Informationen und Strukturen des Anwendungsgebiets, ist oft die *Assoziationsklasse* nützlich: sie beinhaltet Attribute, die weder zu der einen noch zu der anderen Klasse der Assoziation gehören, sondern die es *wegen* der Assoziation gibt. Ein Beispiel:



Vererbung

Gibt es mehrere Klassen mit übereinstimmenden Attributen und Methoden, so kann man sie in eine Vererbungsbeziehung bringen: die *Oberklasse* vererbt ihre Attribute und Methoden an die *Unterklasse*; die Unterklasse kann *weitere* Attribute und Methoden definieren und – wenn es durch die Oberklasse erlaubt ist – auch Methoden *überschreiben*.

Die Vererbung wird in der UML so dargestellt:



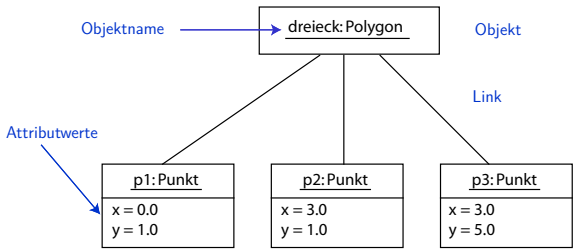
Wichtig ist es zu bemerken, dass die Diagramme etwas über die Komplexität der Sache hinwegtäuschen können. Tatsächlich sind die Konzepte

Assoziation und *Vererbung* orthogonal zueinander: Assoziationen werden auch vererbt, obgleich man das im Diagramm vielleicht nicht auf den ersten Blick sieht. Hilfreich ist, sich das Diagramm in drei Dimensionen vorzustellen, dann sieht man, wie das `Girokonto` an die Stelle des `Kontos` tritt, also eine Beziehung zum `Kunden` hat!

Objektdiagramm

Das Objektdiagramm stellt einen Schnappschuss dar, wie er sich aus einem Klassendiagramm ergeben könnte. Es ist exemplarisch und kann verdeutlichen, wie das Klassendiagramm gemeint ist. Oft ist es auch hilfreich, zunächst beispielhafte Objektdiagramme zu zeichnen, um eine Situation zu überblicken und dann daraus das Klassendiagramm zu entwickeln.

Objekt



Ein Objekt wird dargestellt durch ein Rechteck, das in seinem ersten Abteil den Objektnamen und den Typ des Objekts enthält. Beides ist unterstrichen, was bedeutet, dass es sich um ein Exemplar, nicht um eine Klasse handelt. Der Objektname ist optional, wird er nicht angegeben, spricht man von einem anonymen Objekt.

Im zweiten (optionalen) Abteil des Rechtecks werden die Atrribute angegeben. Da es sich um ein Exemplar handelt, haben sie konkrete Werte. Wie stets in der UML müssen diese Angaben nicht vollständig sein.

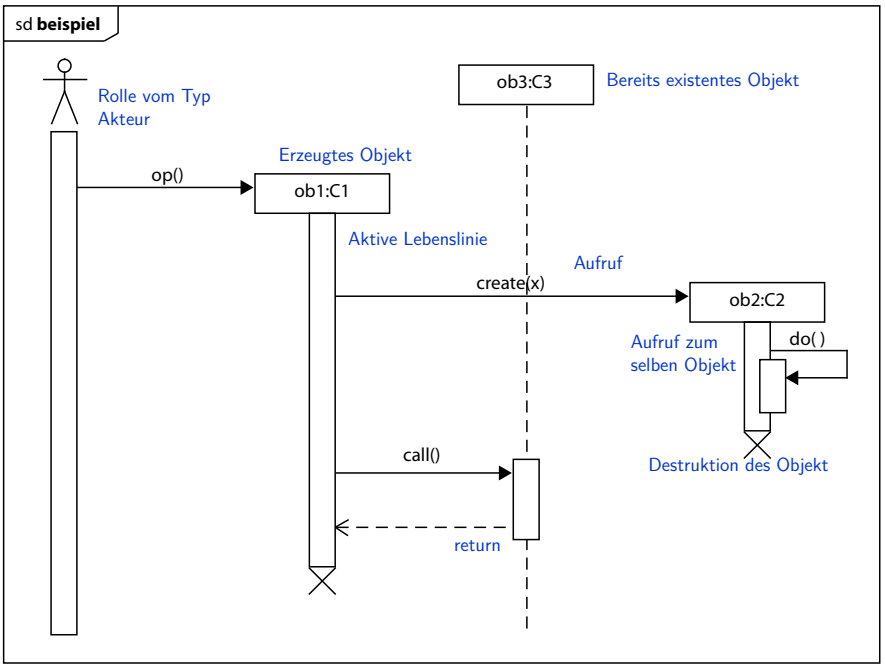
Objektbeziehung

Objektbeziehungen verbinden Objekte, sie sind die Exemplare der Assoziationen im Klassendiagramm. Eine Objektbeziehung wird auch *Link* genannt.

Im obigen Beispiel müsste im zugehörigen Klassendiagramm eine Assoziation zwischen der Klasse Polygon und der Klasse Punkt bestehen. Was sind die Multiplizitätem in diesem Klassendiagramm?

Manchmal sieht man Objektdiagramm, in denen bei den Objektbeziehungen Multiplizitäten angegeben sind. Weshalb ist das falsch?

Sequenzdiagramm



Das Sequenzdiagramm zeigt exemplarisch die Interaktion zwischen Objekten in ihrem zeitlichen Verlauf.

Aufbau

In der linken oberen Ecke erhält das Sequenzdiagramm die Kennzeichnung „sd“ gefolgt vom Namen der im Diagramm beschriebenen Interaktion von Objekten.

Das Diagramm hat zwei Dimensionen: horizontal werden die Objekte angeordnet, in vertikaler Richtung verläuft die Zeit.

Sind beteiligte Objekte bereits zum Zeitpunkt des Beginns der dargestellten Interaktion vorhanden, werden sie ganz oben dargestellt. Objekte können aber auch während der Interaktion erzeugt werden, dann wird die Methode dargestellt, die sie erzeugt und die Destruktion eines Objekts kann durch ein Kreuz angezeigt werden.

Man mag sich fragen, weshalb die Objekte im Sequenzdiagramm nicht unterstrichen dargestellt werden, wie dies im Objektdiagramm der Fall

ist (und wie es in den Sequenzdiagrammen der UML 1.x auch war). In der UML 2.0 werden die Objekte als Rollen dargestellt – ein subtiler Unterschied zum Objekt: „role = A constituent element of a structured classifier that represents the appearance of an instance ...“ [UML Referenz]. Die Vorstellung ist also, dass es sich gewissermaßen um Repräsentanten von Objekten und nicht um die Objekte selbst handelt, die im Sequenzdiagramm dargestellt werden. Das liegt daran, dass in der UML 2 nicht nur Objekte als Teilnehmer im Sequenzdiagramm auftreten können, sondern auch andere Elemente der UML.

Zeitlinie

Jedes Objekt ist in einer eigenen Spalte dargestellt, in der nach unten die Zeitlinie (auch Lebenslinie genannt) verläuft. Sie stellt also dar, dass das Objekt existiert und an der Interaktion teilnehmen kann.

[Nimmt man die Spezifikation genau, dann *ist* die Zeitlinie die Darstellung des Teilnehmers an der Interaktion. Der Name im Kopf der Zeitlinie bezeichnet also zugleich den Teilnehmer und die Zeitlinie.]

Methodenaufruf und Fokus

Pfeile zwischen den Zeitlinien der Objekte bedeuten einen Methodenaufruf. Die Vorstellung ist, dass ein Objekt dem anderen ein Nachricht in Richtung des Pfeils sendet, was eine Methode des aufgerufenen Objekts auslöst. An den Pfeil schreibt man den Namen der Methoden mitsamt eventuell übergebenen Parametern (z.B. andere Objekte im aktuellen Diagramm). Man beachte, dass die aufgerufene Methode zu dem Objekt gehört, auf das die Pfeilspitze zeigt.

In Sequenzdiagrammen können sowohl synchrone als auch asynchrone Aufrufe dargestellt werden. In unserem Beispiel kommen nur synchrone Aufrufe vor. Das bedeutet, dass der Fokus der dargestellten Interaktion mit dem Aufruf wandert. Der Fokus wird durch einen Balken auf der Lebenslinie dargestellt.

Wenn man von rechts her diese Balken aufeinander projeziert, dann entspricht dies genau der Aufrufstruktur des Teilprogramms, das im Sequenzdiagramm dargestellt wird.

Zustandsdiagramm

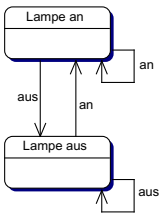
Die Dynamik von Objekten bzw. Systemen

Objekte haben gekapselte Daten und Methoden, mit denen sie verwendet werden können. Man kann bei zustandsorientierten Objekten den aktuellen Wert der Attribute des Objekts (in der Sicht der Abstraktion!) als den *Zustand* des Objekts betrachten. Eine Methode, die den Zustand des Objekts verändert, kann man als *Ereignis* auffassen, auf das das Objekt durch die Zustandsänderung reagiert.

Diese Sichtweise kann man nicht nur für einzelne (durch Klassen beschriebene) Objekte haben, sondern auch für ganze System oder Subsysteme. Sie eignet sich hervorragend, um die *Dynamik*, also die Zustandänderung in Reaktion auf Ereignisse, zu beschreiben.

Wir gegen von folgenden Eigenschaften von Objekten (bzw. Systemen) aus:

- Objekte haben endlich viele diskrete *Zustände*.
- Objekte können auf *Ereignisse* reagieren oder selbst Ereignisse für andere Objekte produzieren.
- Das Verhalten eines Objekts hängt von seinem aktuellen Zustand ab.
- Der aktuelle Zustand hängt von vorhergehenden Ereignissen ab: das Objekt hat ein Gedächtnis.



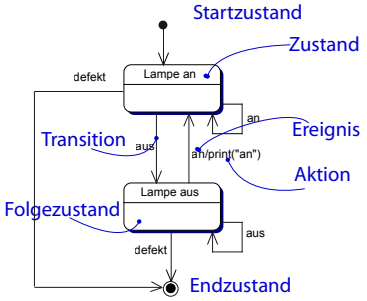
Die Grundidee ist also:

Zustand Definierbare, eindeutige Situation, in dem sich das Objekt befindet und die eine gewisse Zeit andauert.

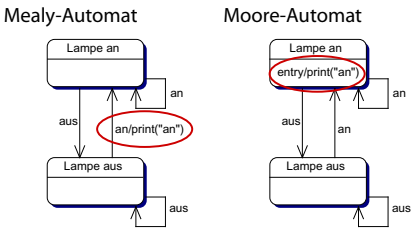
Transition Übergang von einem Zustand in einen anderen; Zustandsübergang.

Ereignis Anstoss für das Objekt, seinen Zustand zu wechseln.

Folgende Abbildung zeigt die Grundelemente eines Zustandsdiagramms. Nebenbei: Das Zustandsdiagramm ist keine Erfindung von UML, sondern stammt von David Harel.[1]

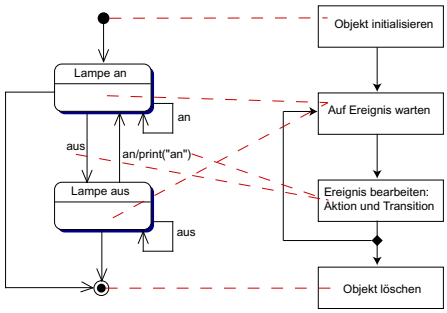


Das Zustandsdiagramm ist die Übertragung des Konzepts der endlichen Automaten aus der Theorie der formalen Sprachen. Dort unterscheidet man zwischen sogenannten Moore- und Mealy-Automaten, wenn es darum geht, dass gewisse *Aktionen* stattfinden. Man kann sich vorstellen, dass solche Aktionen *in* Zuständen stattfinden und eventuelle sogar solange andauern, bis der Zustand durch ein Ereignis verlassen wird – oder das Aktionen ausgeführt werden, wenn ein Ereignis auftritt und eine Transition auslöst. Im Zustandsdiagramm der UML sind beide Möglichkeiten erlaubt, auch der Mix beider Konzepte.



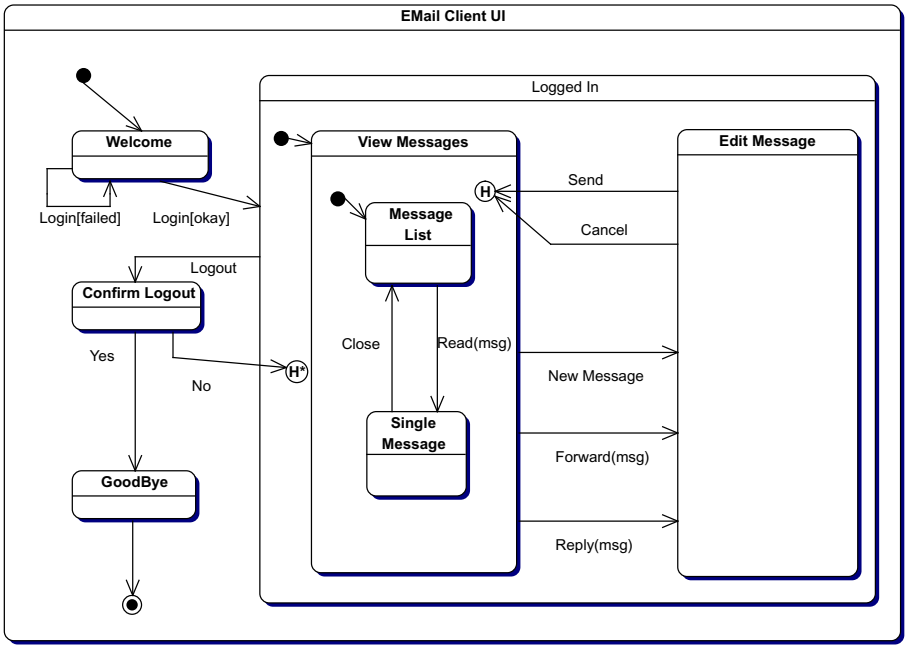
In der Regel ergibt sich aus einem Zustandsdiagramm recht direkt, wie die Dynamik eines Objekts, bzw. eines Systems zu implementieren ist. (Für die Implementierung eines Zustandsautomaten gibt es verschiedene Muster, die wir jedoch hier nicht näher behandeln können. Ein Beispiel für eine Umsetzung folgt.)

[1] D. Harel, „Statecharts: A Visual Formalism for Complex Systems“, *Sci. Comput. Programming* 8 (1987), 231-274.



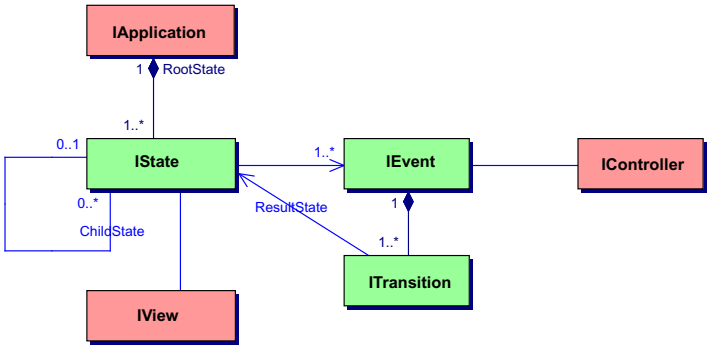
Ein Beispiel soll den Einsatz von Zustandsdiagrammen in der Softwareentwicklung illustrieren. Das Beispiel eines einfachen webbasierten E-Mail-Clients stammt von Brían O'Byrne[2]

Folgende Abbildung zeigt die verschiedenen Webseiten des E-Mail-Client als Zustände eines Zustandsdiagramms und die Navigation in der Anwendung als Transitionen des Zustandsautomaten.



Ein solches Zustandsdiagramm lässt sich direkt in eine Implementierung übertragen, die folgende Klassenstruktur als Basis verwendet. Dabei sind die grünen Klassen vorgegeben und die Umsetzung des Automaten geschieht durch die Implementierung der benötigten roten Interfaces.

[2]Brían O'Byrne: „State Machines & User Interfaces", *Dr. Dobb's Journal*, Januar 2003



Weitere Lektuüre zum Thema:

Practical UML: A Hands-On Introduction for Developers - by Randy Miller
http://edn.embarcadero.com/article/31863

Bodo Igler, Nadja Krümmel, Malte Ried, Burkhardt Renz: *Kurzanleitung UML.* Institut für SoftwareArchitektur. http://homepages.thm.de/~hg11260/mat/uml.pdf

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.0 – 10. April 2012