Objekt-Orientierte-Analyse

SWE1

Inhalt

Artikel

Referenzen

Artikellizenzen

Objektorientierte Analyse und Design

Objektorientierte Analyse und Design (**OOAD**) sind objektorientierte Varianten der zwei allgemeinen Phasen Definition (Objektorientierte Analyse) und Softwarearchitektur (Objektorientiertes Design) im Entwicklungsprozess eines Softwaresystems.

In der *Analyse* geht es darum, die Anforderungen zu erfassen und zu beschreiben, die das zu entwickelnde Softwaresystem erfüllen soll. Stark vereinfacht ausgedrückt sucht und sammelt man in dieser Phase alle Fakten, stellt diese dar und überprüft sie. Dies geschieht oft in Form eines textuellen Pflichtenheftes oder der Software Requirements Specification. Das darauf aufbauende Objektorientierte Analysemodell (OOA-Modell) ist eine fachliche Beschreibung mit objektorientierten Konzepten, oft mit Elementen der Unified Modeling Language (*UML*) notiert. Ziel der Analyse ist ein allgemeines Produktmodell. Implementierungsspezifische technische Details sind noch nicht Betrachtungsgegenstand. Das OOA-Modell kann ein statisches und/oder ein dynamisches Teilmodell enthalten. Im statischen Teil werden Attribute, Vererbungsbeziehungen zwischen Klassen, Assoziationen untereinander und Paketstrukturen betrachtet. Im dynamischen Teil werden mittels Geschäftsprozessdiagrammen, Zustandsautomaten sowie Szenarios die Operationen und Botschaften zwischen Klassen modelliert.

Ergebnis der Analyse sind verschiedene Artefakte, wie Diagramme und Darstellungen von Kontrollstrukturen.

- Entity-Relationship-Modell
- Klassendiagramm
- Pseudocode
- Zustandsautomat
- Kollaborationsdiagramm
- Sequenzdiagramm
- Geschäftsprozessdiagramm (Use-Case-Diagramm/Anwendungsfalldiagramm)

Das Modell kann auch einen Prototypen der Benutzerschnittstelle und eine erste Version des Benutzerhandbuchs enthalten.

Beim *objektorientierten Design* wird das in der Analyse erstellte Domänenmodell weiterentwickelt und darauf aufbauend ein Systementwurf erstellt. Dabei wird das allgemeine Modell in eine konkrete Softwarearchitektur umgeformt, die Informationen über technische Umsetzungsdetails enthält und direkt als Vorlage für die Implementierung in einer Programmiersprache dient.

Ein bekanntes Werkzeug für Analyse und Design ist die *Unified Modeling Language* (UML). Mit Hilfe dieses Modellierungsverfahrens können große Teile der Analyse und des Designs in standardisierter Form beschrieben werden.

Literatur

- Peter Coad, Edward Yourdon: *Objekt-orientierte Analyse.* Prentice Hall Verlag, München 1994, ISBN 3-930436-07-8.
- Peter Coad, Edward Yourdon: *OOD Objektorientiertes Design.* Prentice Hall Verlag, München 1994, ISBN 3-930436-09-4.
- Shlaer, Mellor: *OOA/RD Object-Oriented Analysis and Recursive Development.*

Siehe auch

- Objektorientierte Programmierung
- Prinzipien Objektorientierten Designs

Weblinks

- Aufgaben der OOA und des OOD [1]
- Objektorientierte Entwurfsmuster der Gang of Four [2]

Referenzen

[1] http://www.oszhdl.be.schule.de/gymnasium/faecher/informatik/ooa-ood/index.htm
[2] http://www.isken.org/DesignPattern/index.html

Entity-Relationship-Modell

Das **Entity-Relationship-Modell**, kurz ER-Modell oder **ERM**, deutsch **Gegenstands-Beziehungs-Modell**, dient dazu, im Rahmen der semantischen Datenmodellierung einen Ausschnitt der realen Welt zu beschreiben. Das ER-Modell besteht aus einer Grafik (ER-Diagramm) und einer Beschreibung der darin verwendeten Elemente, wobei Dateninhalte (d. h. die Bedeutung oder Semantik der Daten) und Datenstrukturen dargestellt werden.

Ein ER-Modell dient sowohl in der konzeptionellen Phase der Anwendungsentwicklung der Verständigung zwischen Anwendern und Entwicklern (dabei wird nur das *Was*, also die Sachlogik, und nicht das *Wie*, also die Technik, behandelt), als auch in der Implementierungsphase als Grundlage für das Design der – meist relationalen – Datenbank.

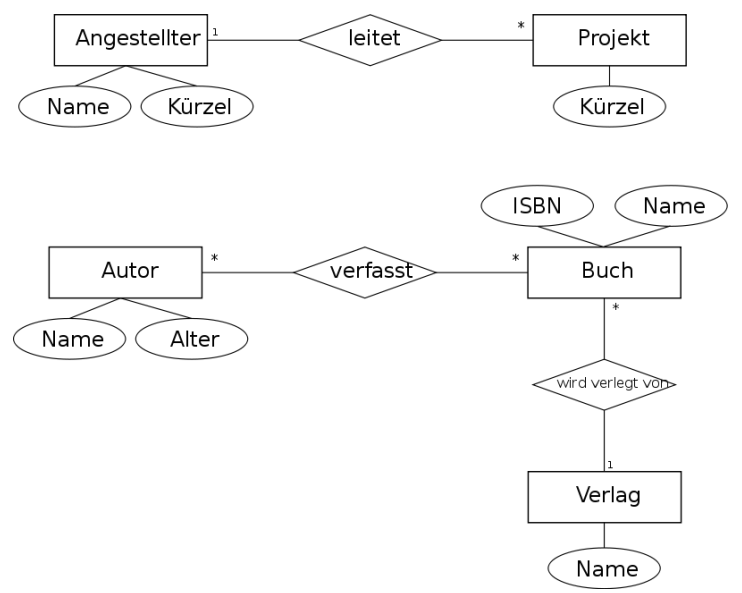
Der Einsatz von ER-Modellen ist der De-facto-Standard für die Datenmodellierung, auch wenn es unterschiedliche grafische Darstellungsformen gibt.

Das ER-Modell wurde 1976 von Peter Chen in seiner Veröffentlichung *The Entity-Relationship Model* vorgestellt. Die Beschreibungsmittel für Generalisierung und Aggregation wurden 1977 von Smith and Smith eingeführt. Danach gab es mehrere Weiterentwicklungen, so Ende der 1980er Jahre durch Wong und Katz.

Begriffe

Grundlage der Entity-Relationship-Modelle ist die Typisierung von Objekten und deren Beziehungen untereinander:

- Entität (Entity): individuell identifizierbares Objekt der Wirklichkeit (zum Beispiel Angestellter „Müller“, Projekt „3232“)
- Entitätstyp: Typisierung gleichartiger Entitäten (zum Beispiel Angestellter, Projekt, Buch, Autor, Verlag)
- Beziehung (Relationship): Verknüpfung zwischen zwei oder mehreren Entitäten (zum Beispiel „Angestellter Müller **leitet** Projekt 3232“)
- Beziehungstyp: Typisierung gleichartiger Beziehungen (zum Beispiel „Angestellter **leitet** Projekt“)



Beispiele für ERMs in Chen-Notation

- reflexive (selbstbezügliche) Beziehung: Beziehung zwischen einzelnen Entitäten ein und desselben Entitätstyps, somit ein Beziehungstyp zwischen dem selben Entitätstyp (zum Beispiel die Baumstruktur einer Aufbauorganisation durch „Organisationseinheit **gliedert sich in** Organisationseinheit“ und die Netzstruktur einer Stückliste durch „Teil **wird verwendet in** Teil“)
- Grad oder Komplexität eines Beziehungstyps: Anzahl der Entitätstypen, die an einem Beziehungstyp beteiligt sind. Die Regel ist der Grad 2 (binärer Beziehungstyp); selten tritt der Grad 3 (ternärer Beziehungstyp) oder ein höherer Grad auf. Ternäre und höhergradige Beziehungstypen lassen sich näherungsweise auf binäre Beziehungstypen durch Einführung eines neuen Entitätstyps, der den ursprünglichen Beziehungstyp beschreibt, zurückführen. Diese Abbildung kann aber verlustbehaftet sein, d.h. es gibt Sachverhalte, die nur durch mehrstellige Beziehungstypen exakt darstellbar sind.
- Kardinalität: Die Kardinalität eines Beziehungstyps legt fest, an wie vielen Beziehungen eine Entität teilnehmen kann (zum Beispiel kann ein Angestellter mehrere Projekte leiten, während ein Projekt von genau einem Angestellten geleitet wird).
- Attribut: Eigenschaft eines Entitätstyps (zum Beispiel Nachname und Geburtsdatum von Entitätstyp Angestellter). Das Attribut oder die Attributkombination eines Entitätstyps, deren Wert(e) die Entität eindeutig beschreiben, d.h. diese identifizieren, heißen identifizierende(s) Attribut(e) (zum Beispiel ist das Attribut Projektnummer identifizierend für den Entitätstyp Projekt). Üblicherweise haben 1:n-Beziehungstypen keine Attribute, da diese immer einem der beteiligten Entitätstypen zugeordnet werden können. Im Falle eines n:m-Beziehungstyps kann aus dem Beziehungstyp ein eigenständiger Entitätstyp mit Beziehungstypen zu den ursprünglich beteiligten Entitätstypen geschaffen werden. Dem neuen Entitätstyp kann dann das Attribut zugeordnet werden (zum Beispiel Attribut Projektbeteiligungsgrad beim n:m-Beziehungstyp „Angestellter arbeitet am Projekt“ zwischen den Entitätstypen Angestellter und Projekt).
- Starker Entitätstyp: Die Identifikation einer Entität ist durch ein oder mehrere Werte von Attributen des gleichen Entitätstyps möglich; so ist z. B. die Auftragsnummer für den Entitätstyp Auftrag identifizierend.
- Schwacher Entitätstyp: Zur Identifikation einer solchen Entität ist ein Attributwert einer anderen mit der schwachen Entität in Beziehung stehenden Entität starken Typs erforderlich; so ist z. B. für die Identifikation des schwachen Entitätstyps Auftragsposition neben der Positionsnummer die Auftragsnummer des anderen starken Entitätstyps Auftrag erforderlich. In Erweiterungen des ER-Modells wie bspw. dem SERM werden Schwacher

Entitätstyp und dazugehöriger Beziehungstyp zu einem sogenannten ER-Typen zusammengezogen, wodurch Diagramme kompakter werden.

Bei der Datenmodellierung wird in den Diskussionen und Beispielen oft mit den konkreten Objekten gearbeitet (Entitäten und Beziehungen). Das Modell selbst besteht aber immer ausschließlich aus Entitätstypen und Beziehungstypen. Vielleicht wird aus diesem Grund oftmals nicht sauber zwischen den Begriffen unterschieden.

Beziehungen mit spezieller Semantik

Die inhaltliche Bedeutung der Beziehungstypen zwischen Entitätstypen kommt im ER-Diagramm lediglich durch einen kurzen Text in der Raute (meistens ein Verb) oder als Beschriftung der Kante zum Ausdruck, wobei es dem Modellierer freigestellt ist, welche Bezeichnung er vergibt. Nun gibt es Beziehungen mit spezieller Semantik, die relativ häufig bei der Modellierung vorkommen. Daher hat man für diese Beziehungstypen spezielle Bezeichner und grafische Symbole definiert. Spezialisierung und Generalisierung sowie Aggregation und Zerlegung sind ergänzende Beschreibungsmittel mit einer speziellen Semantik. Mit diesen beiden speziellen Beziehungen kann die Realwelt verfeinert oder vergröbert modelliert werden. Mit fest definierten Namen und speziellen grafischen Symbolen wird gezeigt, dass es sich um semantisch vorbesetzte Beziehungen handelt.

Spezialisierung und Generalisierung mittels *is-a*-Beziehung

Bei der Spezialisierung wird ein Entitätstyp als Teilmenge eines anderen Entitätstyps deklariert, wobei sich die Teilmenge (spezialisierte Menge) durch besondere Eigenschaften (spezielle Attribute und/oder Beziehungen) gegenüber der übergeordneten (generalisierten Menge) auszeichnet. Da es sich bei einem Einzelobjekt einer spezialisierten Menge um dasselbe Einzelobjekt der generalisierten Menge handelt, gelten alle Eigenschaften – insbesondere die Identifikation – und alle Beziehungen des generalisierten Einzelobjektes auch für das spezialisierte Einzelobjekt.
Die Beziehung Spezialisierung/Generalisierung wird durch *is-a*/*can-be* (‚ist ein'/‚kann ein … sein') beschrieben. Für *is-a* wird gelegentlich auch *a-kind-of* (‚eine Art …') benutzt. Es handelt sich hierbei um eine 1:c-Beziehung.

Beispiel zur *is-a*-Beziehung: Dackel is-a Hund
und in anderer Leserichtung: Hund can-be Dackel

Die hier beschriebene *is-a*-Beziehung zwischen Mengen darf nicht mit der *is-element-of*-Beziehung, der Zugehörigkeit eines Elements zu einer Menge, verwechselt werden, für die gelegentlich auch die Schreibweise *is-a* verwendet wird, wie z. B. Waldi is-a Dackel.

Die Spezialisierung erhält man durch Aufteilung, während die Generalisierung durch Zusammenführen von gleichen Einzelobjekten mit gemeinsamen Eigenschaften und Beziehungen, die in verschiedenen Entitätstypen vorkommen, in einem neuen Entitätstyp begründet ist. So können z. B. Kunden und Lieferanten zusätzlich zu Geschäftspartnern zusammengeführt werden, da Name, Anschrift, Bankverbindung etc. sowohl bei den Kunden als auch bei den Lieferanten vorkommen.

Die Visualisierung von Spezialisierung und Generalisierung ist im ursprünglichen ERM Diagramm nicht vorgesehen, aber in Erweiterungen wie z. B. dem SERM.

Aggregation und Zerlegung mittels *is-part-of*-Beziehung

Werden mehrere Einzelobjekte (z. B. Person und Hotel) zu einem eigenständigen Einzelobjekt (z. B. Reservierung) zusammengefasst, dann spricht man von Aggregation. Dabei wird das übergeordnet eigenständige Ganze Aggregat genannt; die Teile, aus denen es sich zusammensetzt, heißen Komponenten. Aggregat und Komponenten werden als Entitätstyp deklariert.

Bei Aggregation/Zerlegung wird zwischen Rollen- und Mengenaggregation unterschieden:
Eine Rollenaggregation liegt vor, wenn es mehrere rollenspezifische Komponenten gibt, diese zu einem Aggregat zusammengefasst werden und es sich um eine 1:c-Beziehung handelt.

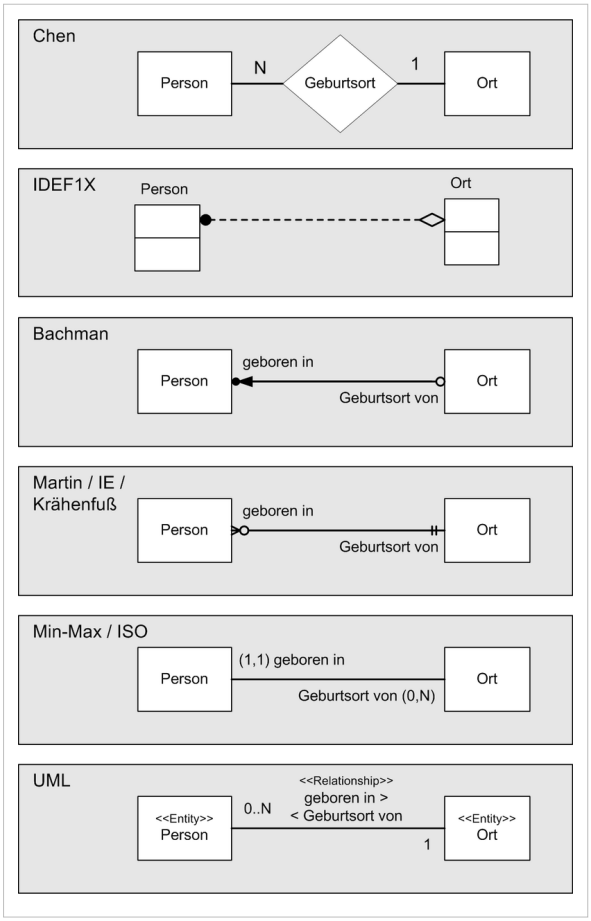
Beispiel zur *is-part-of*-Beziehung: Fußballmannschaft is-part-of Fußballspiel und Spielort is-part-of Fußballspiel und in anderer Leserichtung: Fußballspiel besteht-aus Fußballmannschaft und Spielort.

Eine Mengenaggregation liegt vor, wenn das Aggregat durch Zusammenfassung von Einzelobjekten aus genau einer Komponente entsteht. Hier liegt eine 1:cN-Beziehung vor.

Beispiel zur Mengenaggregation: Fußballspieler is-part-of Fußballmannschaft
und in anderer Leserichtung: Fußballmannschaft besteht aus (mehreren, N) Fußballspielern.

ER-Diagramme

Die grafische Darstellung von Entitätstypen und Beziehungstypen wird *Entity-Relationship-Diagramm* (ERD) oder *ER-Diagramm* genannt. Es sind unterschiedliche Darstellungsformen in Gebrauch. Für den Entitätstyp wird meistens ein Rechteck verwendet, der Beziehungstyp meistens in Form einer Verbindungslinie mit besonderen Linienenden oder Beschriftungen, die die Kardinalitäten des Beziehungstyps darstellen.



Es gibt heute eine Vielzahl unterschiedlicher *Notationen*, die sich unter anderem in Klarheit, Umfang der grafischen Sprache, Unterstützung durch Standards und Werkzeuge unterscheiden. Im Folgenden finden sich einige wichtige Beispiele, die vor allem deutlich machen, dass bei allen grafischen Unterschieden die Kernaussage der ER-Diagramme nahezu identisch ist.

Von besonderer – zum Teil historischer – Bedeutung sind unter anderem:

- die Chen-Notation von Peter Chen, dem Entwickler der ER-Diagramme, 1976;
- die IDEF1X als langjähriger De-Facto-Standard bei US-amerikanischen Behörden;
- die Bachman-Notation von Charles Bachman als weit verbreitete Werkzeug-Diagramm-Sprache;
- die Martin-Notation (Krähenfuß-Notation) als weit verbreitete Werkzeug-Diagramm-Sprache (Information Engineering);
- die (min, max)-Notation von Jean-Raymond Abrial, 1974.
- UML als Standard, den selbst ISO in eigenen Normen als Ersatz für ER-Diagramme verwendet. Attribute (im Schaubild nicht zu sehen) können als Klassenattribute dargestellt werden; Relationship-Attribute hingegen

werden mit Hilfe von Assoziationsklassen modelliert.

Alle nebenstehenden Notationen drücken auf ihre Art den folgenden Sachverhalt aus:

- *Eine Person ist in maximal einem Ort geboren. Ein Ort ist Geburtsort von beliebig vielen Personen.*
- *Ein Ort kann ein Geburtsort sein, muss es aber nicht sein.*

Bis auf das Chen-Diagramm wird diese Aussage ergänzt um:

- *Eine Person muss in einem Ort geboren sein, oder ist in genau einem Ort geboren.*

Die (min, max)-Notation unterscheidet sich grundlegend von den anderen Notationsformen im Hinblick auf die Bestimmung der Kardinalität und den Ort, an dem die Häufigkeitsangabe im ER-Diagramm vorgenommen wird. Bei allen anderen Notationen wird die Kardinalität eines Beziehungstyps dadurch bestimmt, dass für eine Entität des einen Entitätstyps nach der Anzahl der möglichen beteiligten *Entitäten* des anderen Entitätstyps gefragt wird. Bei der Min-Max-Notation hingegen ist die Kardinalität anders definiert. Hierbei wird für jeden der an einem Beziehungstyp beteiligten Entitätstyp nach der kleinst- und größtmöglichen Anzahl der *Beziehungen* gefragt, an denen eine Entität des jeweiligen Entitätstyps beteiligt ist. Das jeweilige Min-Max-Ergebnis wird bei dem Entitätstyp notiert, für den die Frage gestellt worden ist.

Der zahlenmäßige Unterschied zwischen Min-Max-Notation und allen anderen Notationen tritt erst bei ternären und höhergradigen Beziehungstypen hervor. Bei binären Beziehungstypen ist der Unterschied lediglich in einer Vertauschung der Kardinalitätsangaben ersichtlich.

Einsatz in der Praxis

Das ER-Modell kann bei der Erstellung von Datenbanken genutzt werden. Hierbei wird mit Hilfe von ER-Modellen zunächst die Konzeption der Datenbank vorgenommen, auf deren Grundlage dann die Implementierung der Datenbank erfolgt. Die Umsetzung der in der Realwelt erkannten Objekte und Beziehungen in ein Datenbankschema erfolgt dabei in mehreren Schritten:

- Erkennen und Zusammenfassen von Objekten zu Entitätstypen durch Abstraktion (z. B.: Die Kollegen Fritz Maier und Paul Lehmann und viele weitere zum Entitätstyp *Angestellter*);
- Erkennen und Zusammenfassen von Beziehungen zwischen je zwei Objekten zu einem Beziehungstyp (Beispiel: Der Angestellte Paul Lehmann leitet das Projekt *Verbesserung des Betriebsklimas*, und der Angestellte Fritz Maier leitet das Projekt *Effizienzsteigerung in der Verwaltung*. Diese Feststellungen führen zum Beziehungstyp „Angestellter leitet Projekt“.);
- Bestimmung der Kardinalitäten, d. h. der Häufigkeit des Auftretens (Wie z. B.: Ein Projekt wird immer von genau einem Angestellten geleitet, und ein Angestellter darf mehrere Projekte leiten).

All dieses lässt sich in einem ER-Modell darstellen.

Weiter sind folgende Schritte notwendig, deren Ergebnis meistens jedoch nicht grafisch dargestellt wird (so z. B. in der obigen Grafik):

- Bestimmung der relevanten Attribute der einzelnen Entitätstypen.
- Markierung bestimmter Attribute eines Entitätstyps als identifizierende Attribute, so genannte Schlüsselattribute.
- Generierung des Schemas einer relationalen Datenbank mit all seinen Tabellen- und zugehörigen Felddefinitionen mit ihren jeweiligen Datentypen.

Überführung in ein relationales Modell

Die Überführung eines Entity-Relationship-Modells in das Relationen-Modell basiert im Wesentlichen auf den folgenden Abbildungen:

- Entitätstyp → Relation
- Beziehungstyp → Fremdschlüssel; im Falle eines n:m-Beziehungstyps → Relation
- Attribut → Attribut.

Die genaue Überführung, die automatisiert werden kann, erfolgt in 7 Schritten:

1. Starke Entitätstypen

Für jeden starken Entitätstyp wird eine Relation *R* mit den Attributen $R = \{a_1, a_2, ..., a_n\} \cup k$ mit *k* als Primärschlüssel und $a_1, a_2, ..., a_n$ als Attribute der Entität erstellt.

2. Schwache Entitätstypen

Für jeden schwachen Entitätstyp wird eine Relation *R* erstellt mit den Attributen $R = \{a_1, a_2, ..., a_n\} \cup \{k\}$ mit dem Fremdschlüssel *k* sowie dem Primärschlüssel $\{k\} \cup \{a_x\}$, wobei $\{a_x\}$ den schwachen Entitätstyp und *k* den starken Entitätstyp identifizieren.

3. 1:1-Beziehungstypen

Für einen 1:1-Beziehungstyp der Entitätstypen *T*, *S* wird eine der beiden Relationen um den Fremdschlüssel für die jeweils andere Relation erweitert.

4. 1:N-Beziehungstypen

Für den 1:N-Beziehungstyp der Entitätstypen *T*, *S* wird die mit der Kardinalität N (bzw. 1 in min-max-Notation) eingehende Relation *T* um den Fremdschlüssel der Relation *S* erweitert.

5. N:M-Beziehungstypen

Für jeden N:M-Beziehungstyp wird eine neue Relation *R* mit den Attributen $R = \{a_1, a_2, ..., a_n\} \cup \{k_T\} \cup \{k_S\}$ mit $\{a_1, a_2, ..., a_n\}$ für die Attribute der Beziehung sowie k_T bzw. k_S für die Primärschlüssel der beteiligten Relationen erstellt.

6. Mehrwertige Attribute

Für jedes mehrwertige Attribut in *T* wird eine Relation *R* mit den Attributen $R = \{k\} \cup \{a_x\}$ mit $\{a_x\}$ als mehrwertiges Attribut und *k* als Fremdschlüssel auf *T* erstellt.

7. n-äre Beziehungstypen

Für jeden Beziehungstyp mit einem Grad $n > 2$ wird eine Relation *R* erstellt mit den Attributen $R = \{k_1, k_2, ..., k_n\} \cup \{a_1, a_2, ...a_m\}$ mit $\{k_1, k_2, ..., k_n\}$ als Fremdschlüssel auf die eingehenden Entitätstypen sowie $\{a_1, a_2, ...a_m\}$ als Attribute des Beziehungstyps. Wenn alle beteiligten Entitytypen mit Kardinalität > 1 eingehen, so ist der Primärschlüssel die Menge aller Fremdschlüssel. In allen anderen Fällen umfasst der Primärschlüssel $n - 1$ Fremdschlüssel, wobei die Fremdschlüssel zu Entitytypen mit Kardinalität > 1 in jedem Fall im Primärschlüssel enthalten sein müssen.

Siehe auch

- Liste von Datenmodellierungswerkzeugen
- Structured Entity Relationship Model, auf ER aufbauend, Methodik führt zu Modell in 3NF, incl. Generalisierung/Spezialisierung, kompaktere Darstellung nach Existenzabhängigkeiten sortiert, daher direktes Ablesen von Einstiegspunkten, Zyklen und Schlüsselvererbung möglich.

Literatur

- Peter Pin-Shan Chen: *The Entity-Relationship Model--Toward a Unified View of Data* [1]. In: ACM Transactions on Database Systems 1/1/1976 ACM-Press ISSN , S. 9–36
- Peter Pin-Shan Chen: *Entity-Relationship Modeling--Historical Events, Future Trends, and Lessons Learned* [2]. In: Software Pioneers: Contributions to Software Engineering, Broy M. and Denert, E. (eds.), Springer-Verlag, Berlin, Lecturing Notes in Computer Sciences, June 2002, pp. 296-310, ISBN 3-540-43081-4
- J.M. Smith, D.C.P. Smith: *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems, Vol. 2, No. 2 (1977), S. 105–133
- J. M. Smith and D. C. P. Smith: *Database Abstraction: Aggregation*, Communications of the ACM, Vol. 20, Nr. 6, pp. 405–413, June 1977
- Ramez Elmasri, Shamkant B. Navathe: *Fundamentals of database systems.* Addison Wesley, ISBN

Referenzen

[1] http://csc.lsu.edu/news/erd.pdf

[2] http://bit.csc.lsu.edu/~chen/pdf/Chen_Pioneers.pdf

Klassendiagramm

Strukturdiagramme der UML
Klassendiagramm
Komponentendiagramm
Kompositionsstrukturdiagramm
Objektdiagramm
Paketdiagramm
Verteilungsdiagramm
Profildiagramm
Verhaltensdiagramme der UML
Aktivitätsdiagramm
Anwendungsfalldiagramm
Interaktionsübersichtsdiagramm
Kommunikationsdiagramm
Sequenzdiagramm
Zeitverlaufsdiagramm
Zustandsdiagramm

Ein **Klassendiagramm** ist ein Strukturdiagramm der Unified Modeling Language (UML) zur grafischen Darstellung (Modellierung) von Klassen, Schnittstellen sowie deren Beziehungen. Eine *Klasse* ist in der Objektorientierung ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von Objekten (*Klassifizierung*). Sie dient dazu Objekte zu abstrahieren. Im Zusammenspiel mit anderen Klassen ermöglichen sie die Modellierung eines abgegrenzten Systems in der Objekt-orientierten Analyse und Entwurf.

Seit den 1990er Jahren werden Klassendiagramme meistens in der Notation der UML dargestellt. Das Klassendiagramm ist eine der 14 Diagrammarten der UML, einer Modellierungssprache für Software und andere Systeme.

Notation in der Unified Modeling Language

Klassen

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse (fett gedruckt) tragen oder zusätzlich auch Attribute, Operationen und Eigenschaften spezifiziert haben. Dabei werden diese drei Rubriken (engl. *compartment*) – Klassenname, Attribute, Operationen und Eigenschaften – jeweils durch eine horizontale Linie getrennt. Wenn die Klasse keine Eigenschaften oder Operationen besitzt, kann die unterste horizontale Linie entfallen. Oberhalb des Klassennamens können Schlüsselwörter (engl. *keyword*) in Guillemets und unterhalb des Klassennamens in geschweiften Klammern zusätzliche Eigenschaften (wie {abstrakt}) stehen.

Die Attribute werden wie folgt spezifiziert:

```
[Sichtbarkeit] [/] name [: Typ] [ Multiplizität ] [= Vorgabewert] [{eigenschaftswert*}]
```

Daraus folgt, dass in der UML ausschließlich der Name eines Attributs angegeben werden muss, und zwar eindeutig innerhalb einer Klasse. Klassenattribute werden unterstrichen. Darüber hinaus sind bei Attributnamen sämtliche Zeichen erlaubt, auch wenn in einigen Programmiersprachen beispielsweise Umlaute verboten sind.

Operationen werden in ähnlicher Art und Weise spezifiziert:

```
[Sichtbarkeit] name [({Parameter})] [: Rückgabetyp] [{eigenschaftswert*}]
```

Zudem wird ein Parameter wie folgt aufgebaut:

```
[Übergaberichtung] name : Typ [ Multiplizität ] [= Vorgabewert] [{eigenschaftswert*}]
```

Die Namensgebung und der Zeichenraum sind hier genauso wie bei den Attributsspezifikationen. Klassenoperationen werden auch hier unterstrichen. Den „Pseudotyp“ void gibt es in der UML nicht, daher muss in einem solchen Fall der Rückgabetyp weggelassen werden. Ansonsten können bei Attributen und Operationen sämtliche primitiven Typen sowie selbst definierte Klassen oder Interfaces als Typ bzw. Rückgabetyp verwendet werden.

Die Sichtbarkeit von Operationen und Attributen wird wie folgt gekennzeichnet:

- „+“ für *public* – (engl. öffentlich), unbeschränkter Zugriff
- „#“ für *protected* – (engl. geschützt), Zugriff nur von der Klasse sowie von Unterklassen (Klassen, die erben)
- „–“ für *private* – (engl. privat), nur die Klasse selbst kann es sehen
- „~“ für *package* - innerhalb des Pakets sichtbar

Mögliche Eigenschaften sind:

ordered

: die Daten werden geordnet zurückgegeben

redefines <Operationsname> (nur bei Operationen)

: diese Operation überschreibt die geerbte Operation <Operationsname>

read-only

auf diese Variable kann nur lesend zugegriffen werden

Die Übergaberichtungen:

in

Der übergebene Parameter wird nur gelesen (Standard, wenn nichts angegeben wurde).

out

Der übergebene Parameter wird beschrieben, ohne ihn vorher zu lesen.

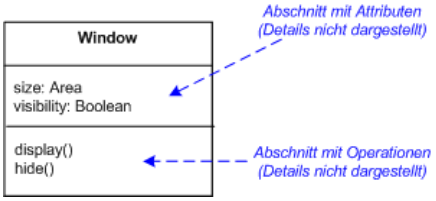
inout

Der übergebene Parameter wird gelesen bzw. verarbeitet und beschrieben, beispielsweise um das Ergebnis zurückzugeben.

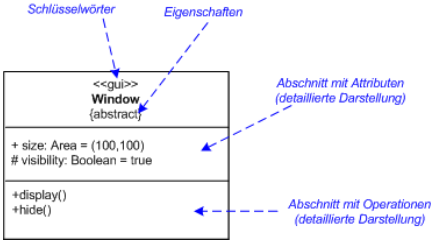
Die folgenden Abbildungen zeigen zwei Varianten der grafischen Notation für eine Klasse. Abhängig davon, ob eine Klasse in einem Klassendiagramm für ein Design- oder für ein Analysemodell gezeichnet wird, können mehr oder weniger Details dargestellt werden.

Einfachste Form der Darstellung für eine Klasse

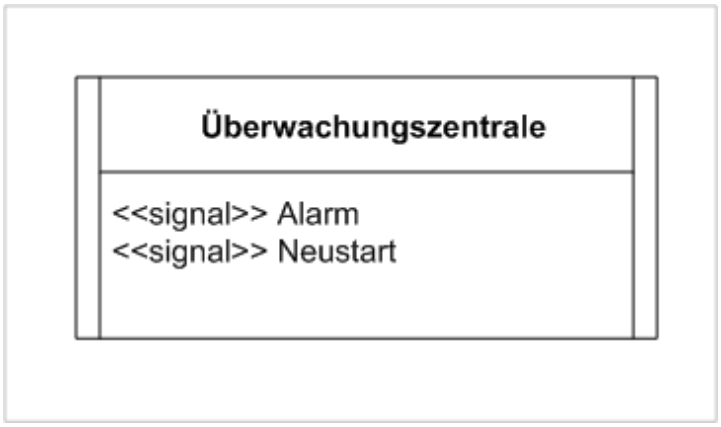
Abstrakte Klassen sind Klassen, von denen keine Instanz angelegt werden kann. Abstrakte Klassen sehen in UML wie normale Klassen aus. Um sie zu unterscheiden, steht unterhalb des Klassennamens das Wort abstract in geschweiften Klammern. Alternativ kann der Klassenname auch kursiv geschrieben werden, wenn dies gut erkennbar ist.



Zusätzliche Darstellung von Attributen

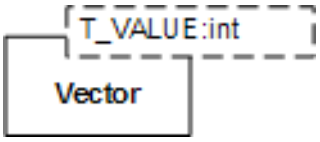


Detaillierte Darstellung einer Klasse



Beispiel einer *aktiven Klasse* mit zwei Signalempfängern

Eine aktive Klasse wird mit einem doppelten linken und rechten Rand gezeichnet.

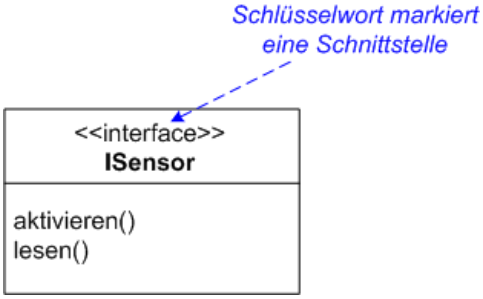


Klassenschablone

Manche Programmiersprachen ermöglichen eine Parametrisierung von Klassenschablonen *(Class Templates)*, um Objekte basierend auf diesen Vorlagenparametern zu erzeugen. Die UML bietet dafür die Notation für *Template Arguments* an. Dabei werden die Vorlagenparameter in einem gestrichelten Rechteck überlappend an die rechte obere Ecke der Klasse eingetragen. Im Beispiel ist eine Klasse „Vector“ mit dem Vorlagenparametertyp „int“ und dem Parameternamen „T_VALUE“ eingetragen.

Schnittstellen

Eine Schnittstelle wird ähnlich wie eine Klasse mit einem Rechteck dargestellt, zur Unterscheidung aber mit dem Schlüsselwort interface gekennzeichnet.

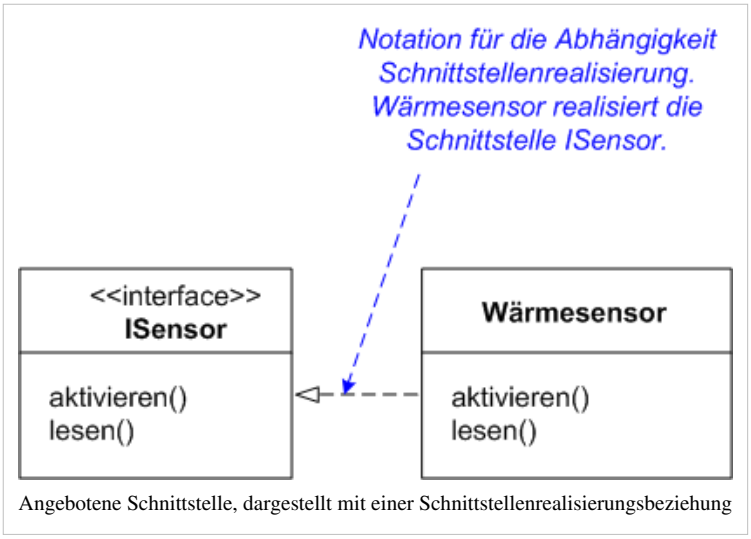


Eine Schnittstelle wird mit dem Schlüsselwort <<interface>> markiert

Wichtige Beziehungen

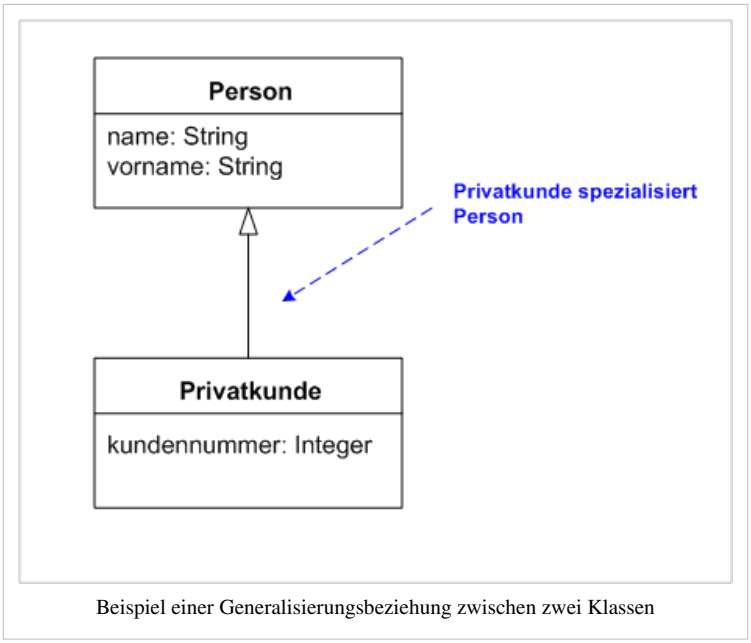
Generalisierung

Eine Generalisierung in der UML ist eine gerichtete Beziehung zwischen einer *generelleren* und einer *spezielleren* Klasse. Exemplare der spezielleren Klasse sind damit auch Exemplare der generelleren Klasse. Konkret bedeutet dies, dass die speziellere Klasse implizit über alle Merkmale (Struktur- und Verhaltensmerkmale) der generelleren Klasse verfügt – *implizit* deshalb, weil diese Merkmale in der spezielleren Klasse nicht explizit deklariert werden. Man sagt, dass die speziellere Klasse sie von der generelleren Klasse „erbt“ oder „ableitet“.



Angebotene Schnittstelle, dargestellt mit einer Schnittstellenrealisierungsbeziehung

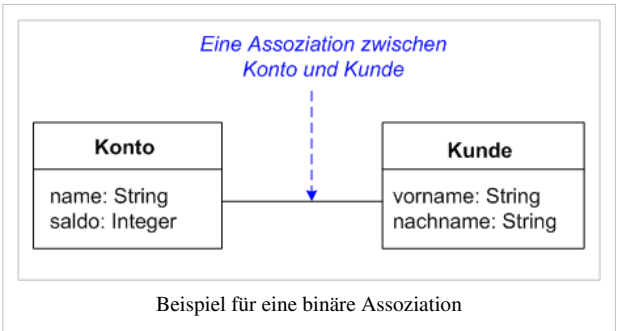
Eine Generalisierung wird als durchgezogene Linie zwischen den beiden beteiligten Classifiern dargestellt. Am Ende mit dem generelleren Classifier wird eine geschlossene, nicht ausgefüllte Pfeilspitze gezeichnet.



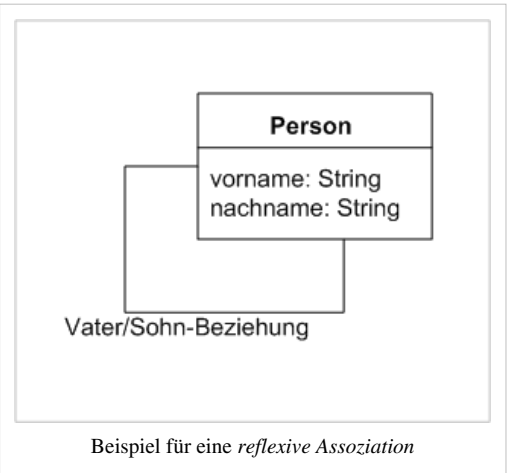
Beispiel einer Generalisierungsbeziehung zwischen zwei Klassen

Assoziation

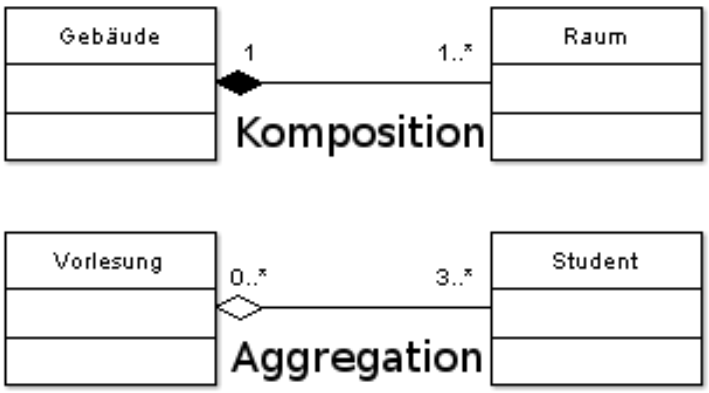
Eine Assoziation beschreibt eine Beziehung zwischen zwei oder mehr Klassen. An den Enden von Assoziationen sind häufig Multiplizitäten vermerkt. Diese drücken aus, wie viele dieser Objekte in Relation zu den anderen Objekten dieser Assoziation stehen.



Beispiel für eine binäre Assoziation



Beispiel für eine *reflexive Assoziation*



Beispiele für *Komposition* und *Aggregation*

Eine Beziehung zwischen Klassen, die relativ häufig modelliert wird, ist die Beziehung zwischen einem *Ganzen* und seinen *Teilen*. Die UML sieht dafür zwei spezielle Assoziationen vor: die *Komposition* und die *Aggregation*.

In der grafischen Darstellung einer *Komposition* dekoriert eine ausgefüllte Raute das Ende, das mit dem Ganzen verbunden ist. Im Fall der *Aggregation* ist es eine nicht ausgefüllte Raute.

Die Komposition ist ein Spezialfall der Aggregation und bildet den Fall ab, bei dem die *Teile* nicht ohne das *Ganze* existieren können (Existenzabhängigkeit).

Formale Semantik

Rumbaugh, Jacobson und Booch fordern eine eher minimal definierte, mengentheoretische Semantikbeschreibung.[1] Demnach ist eine Konfiguration (englisch *snapshot*) σ eines UML-Klassendiagrammes eine Menge von Objekten der in dem Diagramm vorhandenen Klassen. Eine Konfiguration ist konsistent, wenn alle in dem Diagramm angegeben Einschränkungen eingehalten werden, wie z. B. Multiplizitäten oder OCL Constraints.

Klassen und Attribute

In jeder Konfiguration wird eine Klasse als Menge ihrer Objekte beschrieben. Wenn $cname$ der Name einer Klasse ist, dann ist $\sigma(cname)$ eine Menge. Diese Menge darf auch leer sein, wenn es kein Objekt gibt.

Wenn $attribn$ ein Attribut vom Typ $typn$ einer Klasse mit dem Klassennamen $cname$ ist, dann ist $\sigma(attribn)$ eine partielle Funktion von der Menge der Objekte $\sigma(cname)$ in die Menge der Objekte des Attributstyps $\sigma(typn)$. Die Funktion muss partiell sein, da sie für (noch) nicht initialisierte Attribute undefiniert ist. Klassenattribute werden genauso behandelt, haben aber die zusätzliche Einschränkung, dass alle Objekte einer Klasse auf dasselbe Objekt des Attributtyps abgebildet werden müssen.
Wurde zusätzlich eine Multiplizität eines Attributes definiert mit dem Intervall I, dann ist $\sigma(attribn)$ eine Relation mit $\sigma(attribn) : \sigma(cname) \times \sigma(typn)$, mit der zusätzlichen Einschränkung, dass für jedes $a \in \sigma(cname)\ Card(\{b|\langle a,b\rangle \in \sigma(attribn)\}) \in I$ gilt.
Falls eine Klasse mit Namen $cname1$ eine Unterklasse von der Klasse mit Namen $cname$ ist, dann gilt: $\sigma(cname1) \subseteq \sigma(cname)$

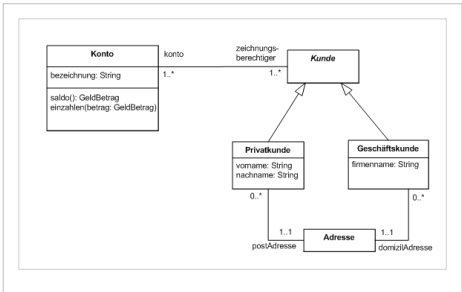
Assoziationen

Eine Assoziation r zwischen Klassen mit den Namen $cname1$ und $cname2$ wird als Relation $\sigma(r)$ zwischen den Mengen der Objekte der Klassen interpretiert, $\sigma(r) : \sigma(cname1) \times \sigma(cname2)$. Die Multiplizitäten müssen in beiden Richtungen wie oben beschrieben behandelt werden. Diese Darstellung erlaubt allerdings keine Behandlung der Rollennamen an den Assoziationsenden. Um dies dennoch zu ermöglichen könnte eine eindeutige Labelfunktion und deren Inverse eingeführt werden.
Bei dieser Art der Betrachtung der Semantik, wird nicht zwischen normalen Assoziationen und deren speziellen Ausprägungen (Aggregation, Komposition) unterschieden.

Operationen

Im Allgemeinen löst eine Operation einen Übergang von einer Konfiguration zu einer anderen aus. Im Falle nicht-deterministischer Operationen gibt es eine Menge von Nachfolge-Konfigurationen. Einen Sonderfall stellen Query-Operationen dar. Da diese keine Seiteneffekte haben dürfen, erfolgt auch kein Zustandsübergang in eine andere Konfiguration. Operationen entsprechen in vielen Programmiersprachen Methoden bzw. Funktionen.

Beispieldiagramm



Literatur

- Heide Balzert: *Lehrbuch der Objektmodellierung – Analyse und Entwurf mit der UML 2*, Elsevier Spektrum Akademischer Verlag, 2005, ISBN 3-8274-1162-9
- Christoph Kecher: *UML 2.0 – Das umfassende Handbuch*, Galileo Computing, 2006, ISBN 3-89842-738-2
- Chris Rupp, Stefan Queins, Barbara Zengler: *UML 2 Glasklar*, Hanser Verlag, 2007, ISBN 978-3-446-41118-0
- James Rumbaugh, Ivar Jacobson, & Grady Booch: *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998, ISBN 978-0201309980

Einzelnachweise

[1] James Rumbaugh, Ivar Jacobson und Grady Booch: *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998, ISBN 978-0201309980.

Pseudocode

Pseudocode ist eine sprachliche Mischung aus natürlicher Sprache, mathematischer Notation und einer höheren Programmiersprache. Wie Flussdiagramme und Nassi-Shneiderman-Diagramme ist auch Pseudocode eine Möglichkeit, Algorithmen darzustellen. Ein Algorithmus wird in Pseudocode einerseits genauer beschrieben als in natürlicher Sprache, anderseits aber noch nicht so detailliert wie durch ein Computerprogramm. Ein Programm in Pseudocode dient ausschließlich dazu, um von Menschen gelesen, nicht aber von einem Computer ausgeführt zu werden. Pseudocode ist also *keine* Programmiersprache.

Für Pseudocode gibt es keine verbindlichen Vorschriften. Jeder kann seine eigene Variante zurechtschneidern. Das Ziel ist, Algorithmen verständlich und klar auszudrücken, ohne auf die Eigenheiten einer Programmiersprache Rücksicht nehmen zu müssen.

Verwendung

Um einen Algorithmus zu verstehen, kann man ihn als Programm untersuchen. Das wird aber erschwert durch die Eigenheiten der Programmiersprache, vor allem ihre Syntax. Zudem haben verschiedene Programmiersprachen unterschiedliche Syntaxen. Jede Formulierung als Programm in einer bestimmten Programmiersprache schließt alle Leser aus, die dieser Sprache nicht mächtig sind. Deshalb formuliert man den Algorithmus zwar ähnlich einem Programm, aber ohne auf eine bestimmte Programmiersprache einzugehen: in Pseudocode.

Pseudocode wird dann eingesetzt, wenn die Funktionsweise eines Algorithmus erklärt werden soll und Einzelheiten der Umsetzung in eine Programmiersprache stören würden. Ein typisches Beispiel sind die Felder, die in Pascal von Eins an indiziert werden, in C dagegen von Null an. In Lehrbüchern werden deshalb Algorithmen gelegentlich in Pseudocode wiedergegeben.

Man kann ein Programm durch Pseudocode spezifizieren. Das sollte allerdings eher vermieden werden, denn die Formulierung als Pseudocode ist bereits eine Programmiertätigkeit, die von der Konzentration auf die Anforderungen ablenkt.[1]

Auch bei der Entwicklung von Algorithmen und der Umformung von Programmen (Programmtransformation, Refactoring) wird Pseudocode eingesetzt.

Aussehen und Stilrichtungen

Pseudocode hat den Anspruch, intuitiv klar zu sein. Geeignete Metaphern aus der Umgangssprache geben einen Verfahrensschritt prägnant wieder, ohne dass dazu eine Erklärung nötig ist, zum Beispiel "durchlaufe das Feld a mit Index i" oder "vertausche die Inhalte der Variablen x und y". Solche Stilmittel verbessern die Übersicht.

Pseudocode kann sich in seinem Stil an eine bestimmte höhere Programmiersprache anlehnen, zum Beispiel an Pascal oder an C.

Im Pascal-Stil werden Schlüsselwörter wie `begin`, `end`, `then`, `do`, `repeat`, `until` benutzt. Im C-Stil werden stattdessen geschweifte Klammern `{,}` gesetzt und das Schlüsselwort `then` wird ausgelassen. Dieser Stil wird oft von Programmierern benutzt, die solche Sprachen verwenden. Beide Stile findet man in Lehrbüchern.

Die Blockstruktur wird gelegentlich auch nur durch Einrücken wiedergegeben.

Eine Liste häufig verwendeter Schlüsselwörter:

Module

- `program` *`Programmname`* `...` `end` *`Programmname`*
- `klasse` *`Klassenname`* `{ ... }`

Fallunterscheidungen

- `if ... then ... else ... end if/exit`
- `wenn ... dann ... sonst ... wenn_ende`
- `falls ... dann ... falls_nicht ... falls_ende`

Schleifen

- `wiederhole ... solange/bis ... wiederhole_ende`
- `while ... do ...`
- `repeat ... until ...`
- `for ... to ... step` *`Schrittweite`* `... next`

Kommentare

- `// kommentar`
- `# kommentar`
- `/* kommentar */`

Definition von Funktionen

- `function() ... begin ... end`
- `funktion() ... start ... ende`

Zusicherungen

- `assert`
- `jetzt gilt`

Beispiele

```
program Name und Kurzbeschreibung
 LiesDatenStruktur()
 LiesDatenInhalt()
 ...
 if DatenUnvollständig then FehlerMelden und exit
 HauptstatistikBerechnen
 ZusammenstellungBerechnen
 Resultate in HTML-Datei schreiben
end program Name
```

Prozedur: euklid
Zweck: Euklidischer Algorithmus zur Berechnung des größten gemeinsamen Teilers
Parameter: natürliche Zahlen m, n
1. **Falls** $m > n$ **, dann** m und n miteinander vertauschen.
2. **Jetzt gilt** $m \leq n$.
3. **Solange** $m > 0$ **wiederhole**
4. **Setze** $n = n - m$.
5. **Falls** $m > n$ **, dann** m und n miteinander vertauschen.
6. **Jetzt gilt** $m \leq n$.

Ergebnis: n.

Siehe auch

- Programmablaufplan
- Jana (Beschreibungssprache)
- Nassi-Shneiderman-Diagramm
- Kontrollstruktur

Einzelnachweise

[1] Johannes Siedersleben (Hrsg.): *Softwaretechnik.* Hanser, München 2003, ISBN 3-446-21843-2, S. 44ff..

Zustandsautomat

Ein **endlicher Automat** (*EA*, auch Zustandsmaschine, englisch *finite state machine (FSM)*) ist ein Modell des Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen.

Ein Automat heißt endlich, wenn die Menge der Zustände, die er annehmen kann (später S genannt), endlich ist. Ein EA ist ein Spezialfall aus der Menge der Automaten. Ein Zustand speichert die Information über die Vergangenheit, d.h. er reflektiert die Änderungen der Eingabe seit dem Systemstart bis zum aktuellen Zeitpunkt. Ein Zustandsübergang zeigt eine Änderung des Zustandes des EA und wird durch logische Bedingungen beschrieben, die erfüllt sein müssen, um den Übergang zu ermöglichen. Eine Aktion ist die Ausgabe des EA, die in einer bestimmten Situation erfolgt. Es gibt vier Typen von Aktionen

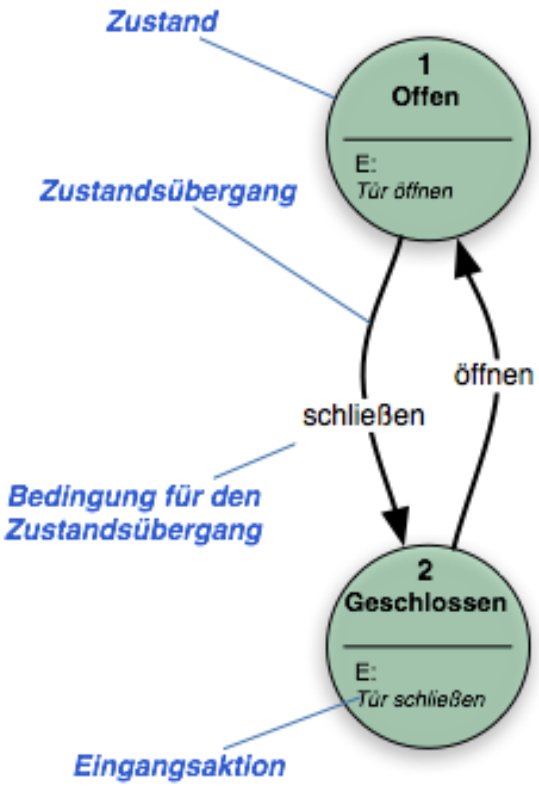


Abb.1 Beispiel eines EA

Eingangsaktion

: Ausgabe wird beim Eintreten in einen Zustand generiert

Ausgangsaktion

: Ausgabe wird beim Verlassen eines Zustandes generiert

Eingabeaktion

: Ausgabe wird abhängig von aktuellem Zustand und Eingabe generiert

Übergangsaktion

: Ausgabe wird abhängig von einem Zustandsübergang generiert

Ein EA kann als Zustandsübergangsdiagramm wie in Abbildung 1 dargestellt werden. Zusätzlich werden mehrere Typen von Übergangstabellen (bzw. Zustandsübergangstabellen) benutzt. Die folgende Tabelle zeigt eine sehr verbreitete Form von Übergangstabellen: die Kombination aus dem aktuellen Zustand (B) und Eingabe (Y) führt zum nächsten Zustand (C). Die komplette Information über die möglichen Aktionen wird mit Hilfe von Fußnoten angegeben. Eine Definition des EA, die auch die volle Ausgabeinformation beinhaltet, ist mit Zustandstabellen

möglich, die für jeden Zustand einzeln definiert werden (siehe auch virtueller EA).

Momentaner Zustand/ Bedingung	Zustand A	Zustand B	Zustand C
Bedingung X	...	...	...
Bedingung Y	...	Zustand C	...
Bedingung Z	...	...	...

|+ Caption | Übergangstabelle

Die Definition des EA wurde ursprünglich in der Automatentheorie eingeführt und später in der Computertechnik übernommen.

Zustandsmaschinen werden hauptsächlich in der Entwicklung digitaler Schaltungen, Modellierung des Applikationsverhaltens (Steuerungen), generell in der Softwaretechnik sowie Wort- und Spracherkennung benutzt.

Klassifizierung

Generell werden zwei Gruppen von EA unterschieden: Akzeptoren und Transduktoren.

Akzeptoren

Sie akzeptieren und erkennen die Eingabe und signalisieren durch ihren Zustand das Ergebnis nach außen. In der Regel werden Symbole (Buchstaben) als Eingabe benutzt. Das Beispiel in der Abbildung 2 zeigt einen EA, der das Wort „gut“ akzeptiert. Akzeptoren werden vorwiegend in der Wort- und Spracherkennung eingesetzt.

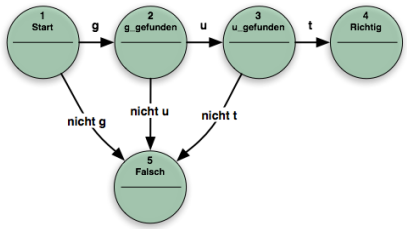


Abb. 2 EA vom Typ Akzeptor: erkennt das Wort "gut"

Transduktoren (Transducer)

Transduktoren generieren Ausgaben in Abhängigkeit von Zustand und Eingabe mit Hilfe von Aktionen. Sie werden vorwiegend für Steuerungsaufgaben eingesetzt, wobei grundsätzlich zwei Typen unterschieden werden:

Moore-Automat

Im Moore-Modell werden nur Eingangsaktionen benutzt, d. h., die Ausgabe (Γ) hängt nur vom Zustand (S) ab ($S \rightarrow \Gamma$). Das Verhalten eines Moore-Automaten ist dadurch, verglichen mit dem Mealy-Modell, einfacher und leichter zu verstehen. Das Beispiel in Abbildung 3 zeigt einen Moore-Automaten, der eine Aufzugtür steuert. Die Zustandsmaschine kennt zwei Befehle, "aufmachen" und "zumachen", die von einem Benutzer eingegeben werden können. Die Eingangsaktion (E:) im Zustand "Aufgehend" startet einen Motor, der die Tür öffnet, und die Eingangsaktion im Zustand "Zugehend" startet den Motor in entgegengesetzter Richtung. Die Eingangsaktionen in den Zuständen "Auf" und "Zu" halten den Motor an. Sie signalisieren außerdem die Situation nach außen (z.B. zu anderen EA).

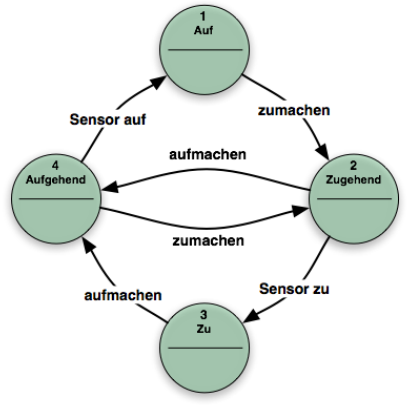


Abb. 3 EA vom Typ Transduktor: Moore-Modell

Mealy-Automat

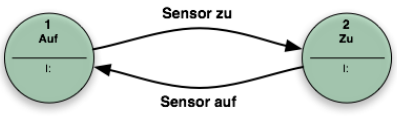


Abb. 4 EA vom Typ Transduktor: Mealy-Modell

Im Mealy-Modell werden Eingabeaktionen benutzt, d. h., die Ausgabe (Γ) hängt von Zustand (S) und Eingabe (Σ) ab ($S \times \Sigma \rightarrow \Gamma$). Der Einsatz von Mealy-Automaten führt oft zu einer Verringerung der Anzahl zu berücksichtigender Zustände. Die Funktion des EA ist dadurch komplexer und oft schwieriger zu verstehen. Das Beispiel in der Abbildung 4 zeigt einen Mealy-EA, der das gleiche Verhalten wie der EA im Moore-Beispiel aufweist. Es gibt zwei Eingabeaktionen (I:): "starte den Motor, um die Tür zu schließen, wenn die Eingabe 'zumachen' erfolgt" und "starte den Motor in entgegengesetzter Richtung, um die Tür zu öffnen, wenn die Eingabe 'aufmachen' erfolgt".

Moore- und Mealy-Automaten sind gleichwertig. Der eine kann in den jeweils anderen überführt werden. In der Praxis werden meistens Mischmodelle benutzt.

Eine weitere Klassifizierung der EA wird durch die Unterscheidung zwischen deterministischen (DEA) und nicht-deterministischen (NEA) Automaten gemacht. In den deterministischen Automaten existiert für jeden Zustand genau ein Übergang für jede mögliche Eingabe. Bei den nicht-deterministischen Automaten kann es keinen oder auch mehr als einen Übergang für die mögliche Eingabe geben.

Ein EA, der nur aus einem Zustand besteht wird als kombinatorischer EA bezeichnet. Er benutzt nur Eingabeaktionen.

Die Logik des EA

Der nächste Zustand und die Ausgabe des EA ist eine Funktion der Eingabe und des aktuellen Zustandes. Abbildung 5 zeigt den Ablauf der Logik.

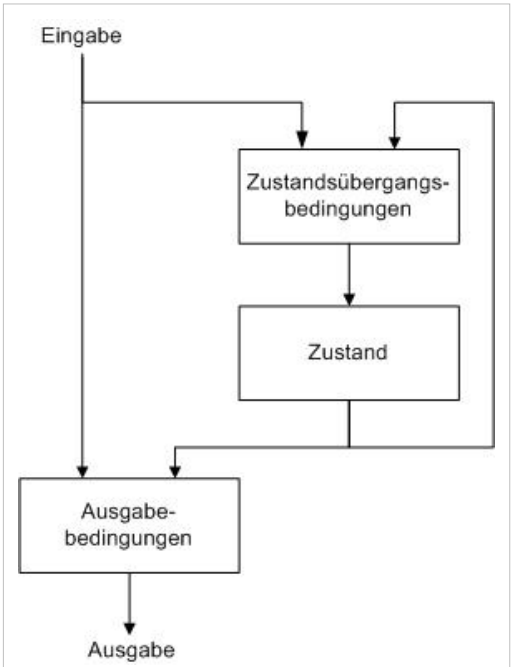


Abb. 5 Die Logik eines EA

Das mathematische Modell

Es gibt unterschiedliche Definitionen, je nach Typ des EA. Ein Akzeptor (oder auch deterministischer Automat) ist ein 5-Tupel (Σ, S, s0, δ, F), wobei:

- Σ ist das Eingabealphabet (eine endliche nicht leere Menge von Symbolen),
- S ist eine endliche nicht leere Menge von Zuständen,
- s0 ist der Anfangszustand und ein Element aus S,
- δ ist die Zustandsübergangsfunktion: $\delta: S \times \Sigma \rightarrow S$,
- F ist die Menge von Endzuständen und eine (möglicherweise leere) Teilmenge von S.

Ein Transduktor ist ein 6-Tupel (Σ, Γ, S, s0, δ, ω), wobei:

- Σ ist das Eingabealphabet (eine endliche nicht leere Menge von Symbolen),
- Γ ist das Ausgabealphabet (eine endliche nicht leere Menge von Symbolen),
- S ist eine endliche nicht leere Menge von Zuständen,
- s0 ist der Anfangszustand und ein Element aus S,
- δ ist die Zustandsübergangsfunktion: $\delta: S \times \Sigma \rightarrow S$,
- ω ist die Ausgabefunktion.

Falls die Ausgabefunktion eine Funktion von Zustand und Eingabealphabet ist (ω: S x $\Sigma \rightarrow \Gamma$), dann handelt es sich um ein Mealy-Modell. Falls die Ausgabefunktion nur vom Zustand abhängt (ω: S $\rightarrow \Gamma$), dann ist es ein Moore-Automat.

Optimierung

Ein EA wird optimiert, indem die Zustandsmaschine mit der geringsten Anzahl von Zuständen gefunden wird, die die gleiche Funktion erfüllt. Dieses Problem kann zum Beispiel mit Hilfe von Färbungsalgorithmen gelöst werden.

Siehe auch: Minimierung eines DEA

Implementierung

Hardware

In digitalen Schaltungen werden EA mit Hilfe von speicherprogrammierbaren Steuerungen, logischen Gattern, Flip-Flops oder Relais gebaut. Eine Hardwareimplementation benötigt normalerweise ein Register, um die Zustandsvariable zu speichern, eine Logikeinheit, die die Zustandsübergänge bestimmt und eine zweite Logikeinheit, die für die Ausgabe verantwortlich ist.

Software

In der Softwareentwicklung werden meist folgende Konzepte verwendet, um Applikationen mit Hilfe von Zustandsmaschinen zu modellieren bzw. implementieren:

- ereignisgesteuerter endlicher Automat
- Virtueller endlicher Automat

Darstellung endlicher Automaten

Die allgemeinen Regeln für das Zeichnen eines Zustandsübergangsdiagramms sind wie folgt:

- Kreise stellen Zustände dar. Im Kreis steht der Name des Zustands.
- Pfeile zwischen Zuständen stellen die Transitionen dar. Auf jedem Pfeil steht, welche Bedingungen den Übergang ermöglichen.

Siehe auch

- Chomsky-Hierarchie

Literatur

- John E. Hopcroft, Jeffrey D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, Bonn 1990, ISBN 3-89319-181-X
- Sander, Stucky, Herschel: *Automaten, Sprachen, Berechenbarkeit.* Teubner, Stuttgart 1992, ISBN 3-519-02937-5
- F. Wagner: *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach, Boca Raton 2006, ISBN 0-8493-8086-3
- Z. Kohavi: *Switching and Finite Automata Theory.* McGraw-Hill, New York 1978, ISBN 0-07-035310-7 (englisch)
- A. Gill: *Introduction to the Theory of Finite-state Machines.* McGraw-Hill, New York 1962 (englisch)
- H.-D. Wuttke, K. Henke: *Schaltsysteme - Eine automatenorientierte Einführung.* Pearson Studium, München 2003, ISBN 3-8273-7035-3

Weblinks

- Formale Sprachen und Abstrakte Automaten [1] – Kurs von Tino Hempel
- Kara – Programmieren mit endlichen Automaten [2] - Kurs: Spielerisch den Marienkäfer "Kara" als EA programmieren und Aufgaben lösen.
- JFLAP [3] - Java-Programm zum Erstellen von Automaten aller Art
- FSMCreator [4] - Java-Programm zum Erstellen von endlichen Automaten

Referenzen

[1] http://tinohempel.de/info/info/ti/index.htm
[2] http://www.swisseduc.ch/informatik/karatojava/kara/
[3] http://www.jflap.org
[4] http://www.fsmcreator.de.gg

Kollaborationsdiagramm

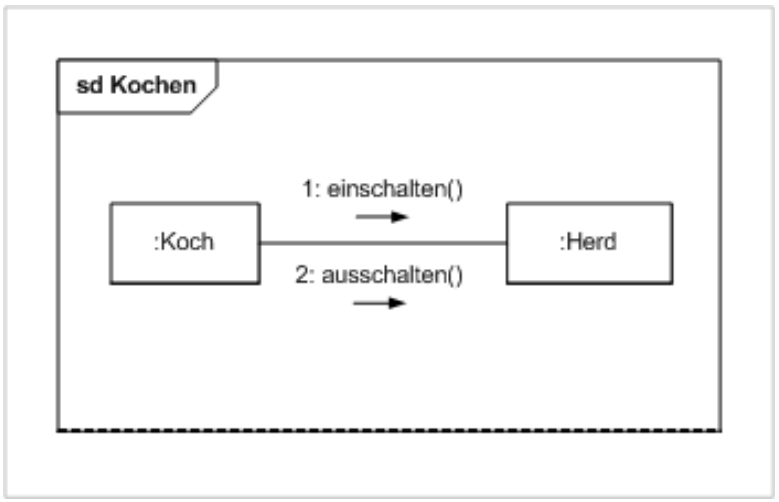
Strukturdiagramme der UML
Klassendiagramm
Komponentendiagramm
Kompositionsstrukturdiagramm
Objektdiagramm
Paketdiagramm
Verteilungsdiagramm
Profildiagramm
Verhaltensdiagramme der UML
Aktivitätsdiagramm
Anwendungsfalldiagramm
Interaktionsübersichtsdiagramm
Kommunikationsdiagramm
Sequenzdiagramm
Zeitverlaufsdiagramm
Zustandsdiagramm

Ein **Kommunikationsdiagramm** (engl. *communication diagram*) ist eine der 14 Diagrammarten in der Unified Modeling Language (UML), einer Modellierungssprache für Software und andere Systeme.

Das Kommunikationsdiagramm ist ein *Verhaltensdiagramm*. Es zeigt eine bestimmte Sicht auf die dynamischen Aspekte des modellierten Systems und stellt Interaktionen grafisch dar, wobei der Austausch von Nachrichten zwischen Ausprägungen mittels Lebenslinien dargestellt wird.

In älteren UML-Versionen war das Kommunikationsdiagramm unter dem Namen **Kollaborationsdiagramm** bekannt.

Notation von Lebenslinien und Nachrichten



Beispiels eines Kommunikationsdiagramms mit zwei Lebenslinien und zwei Nachrichten

Die Abbildung links zeigt ein Beispiel eines Kommunikationsdiagramms mit einem Kopf- und einem Inhaltsbereich. Das optionale Schlüsselwort im Kopfbereich ist bei einem Kommunikationsdiagramm wie bei jedem Interaktionsdiagramm **sd** oder **interaction**.

Ähnlich wie in einem Sequenzdiagramm werden in einem Kommunikationsdiagramm Lebenslinien als Rechtecke dargestellt. Dieses Symbol „Lebenslinie“ zu nennen mag etwas seltsam erscheinen, denn im Kommunikationsdiagramm wird im Unterschied zum Sequenzdiagramm die gestrichelte Linie, die für die Zeitachse beim Austausch von Nachrichten steht, nicht angezeigt. Eine Nachricht wird als kurzer Pfeil gezeichnet. Die Richtung des Pfeils zeigt vom Sender zum Empfänger der Nachricht. Der Pfeil ist beschriftet mit einer Sequenznummer und einer Signatur der Nachricht, zum Beispiel der Signatur der Operation, falls es sich bei der Nachricht um einen synchronen Aufruf einer Operation handelt.

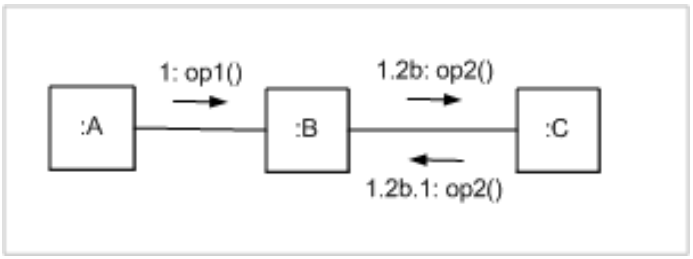
Jeder Verbindung im Kommunikationsdiagramm muss eine Assoziation im Klassendiagramm gegenüberstehen.

Zeitliche Ordnung der Nachrichten

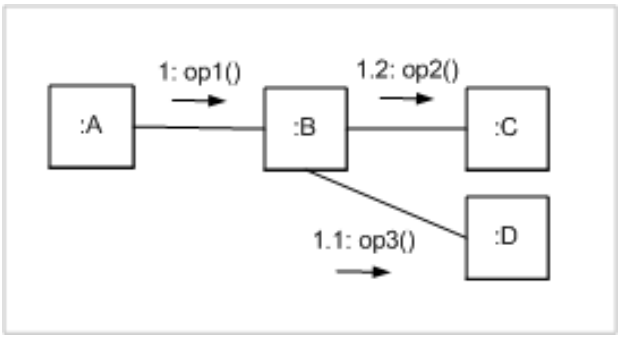
Die zeitliche Abfolge von Nachrichten wird in Kommunikationsdiagrammen mit *Sequenzausdrücken* modelliert. Jeder modellierten Nachricht ist ein Sequenzausdruck zugeordnet, aus dem hervorgeht, was die Vorgängernachrichten sind und welche Nachrichten allenfalls parallel ablaufen.

Ein Sequenzausdruck besteht aus einer Folge von mit einem Punkt (.) getrennten *Sequenztermen*, gefolgt von einem Doppelpunkt (:). Sequenzterme sind Zahlen, zum Beispiel 1, 10 oder 22, die optional von einem Kleinbuchstaben, zum Beispiel a oder b gefolgt werden. Folgende Zeichenfolgen sind also gültige Sequenzausdrücke: 1, 1.7, 1a.6, 1a.6c.2.

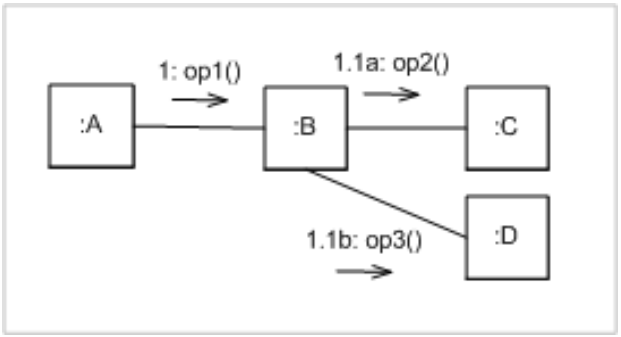
Die Bedeutung von Sequenzausdrücken ist durch drei Regeln gegeben.



Erstens gibt die Folge der Terme im Sequenzausdruck die Verschachtelungstiefe in einer Meldung an. Eine Nachricht mit dem Sequenzausdruck 1.2b.6 wird also von einer Ausprägung gesendet, nachdem diese eine Nachricht mit dem Sequenzausdruck 1.2b erhalten hat. In der Abbildung links folgt die Nachricht mit dem Sequenzausdruck 1.2b.1 auf die Nachricht mit dem Sequenzausdruck 1.2b, diese wiederum auf die Nachricht mit dem Sequenzausdruck 1.



Zweitens geben rein numerische Terme auf der gleichen Verschachtelungstiefe eine sequentielle Reihenfolge an. Die Abbildung links stellt also eine Interaktion dar, in der die Lebenslinie :B die Nachricht mit dem Sequenzausdruck 1.1 vor der Nachricht mit dem Sequenzausdruck 1.2 versendet.



Drittens bezeichnen Terme mit einem Buchstaben am Schluss Nachrichten, die parallel versendet werden. Im Diagramm links erkennt man also, dass die Nachricht mit dem Sequenzausdruck 1.1a und die Nachricht mit dem Sequenzausdruck 1.1b gleichzeitig versendet werden.

Siehe auch

- UML (Unified Modeling Language)

Literatur

- Christoph Kecher: "UML 2.0 - Das umfassende Handbuch" Galileo Computing, 2006, ISBN 3-89842-738-2
- Heide Balzert: "Lehrbuch der Objektmodellierung - Analyse und Entwurf mit der UML 2" Elsevier Spektrum Akademischer Verlag, 2005, ISBN 3-8274-1162-9

Sequenzdiagramm

Strukturdiagramme der UML
Klassendiagramm
Komponentendiagramm
Kompositionsstrukturdiagramm
Objektdiagramm
Paketdiagramm
Verteilungsdiagramm
Profildiagramm
Verhaltensdiagramme der UML
Aktivitätsdiagramm
Anwendungsfalldiagramm
Interaktionsübersichtsdiagramm
Kommunikationsdiagramm
Sequenzdiagramm
Zeitverlaufsdiagramm
Zustandsdiagramm

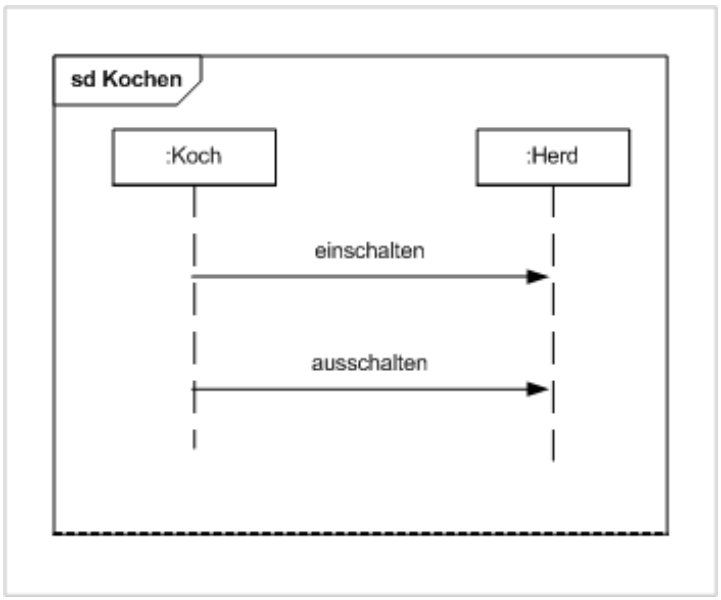
Ein **Sequenzdiagramm** (engl. *sequence diagram*) ist eine der 14 Diagrammarten in der Unified Modeling Language (UML), einer Modellierungssprache für Software und andere Systeme.

Das Sequenzdiagramm ist ein *Verhaltensdiagramm*, genauer eines der vier *Interaktionsdiagramme*. Es zeigt eine bestimmte Sicht auf die dynamischen Aspekte des modellierten Systems. Ein Sequenzdiagramm ist eine grafische Darstellung einer Interaktion und beschreibt den Austausch von Nachrichten zwischen Ausprägungen mittels Lebenslinien.

Sequenzdiagramme der UML2 sind nahe verwandt mit Message Sequence Charts (MSC), einem Standard der ITU-T (International Telecommunication Union - Telecommunication Standardization Sector).

Ein Sequenzdiagramm stellt in der Regel *einen* Weg durch einen Entscheidungsbaum innerhalb eines Systemablaufes dar. Sollen Übersichten mit allen Entscheidungsmöglichkeiten entwickelt werden, so müsste hierzu für jeden möglichen Ablauf ein eigenständiges Sequenzdiagramm modelliert werden; deshalb eignet sich hierfür eher das Aktivitätsdiagramm oder Zustandsdiagramm.

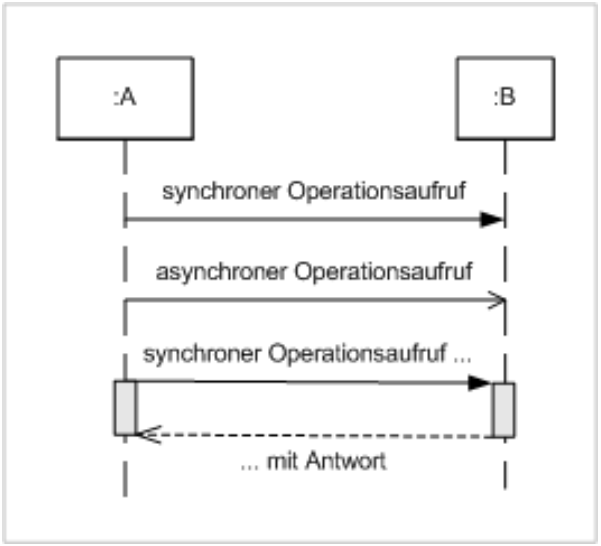
Notation von Lebenslinien und Nachrichten



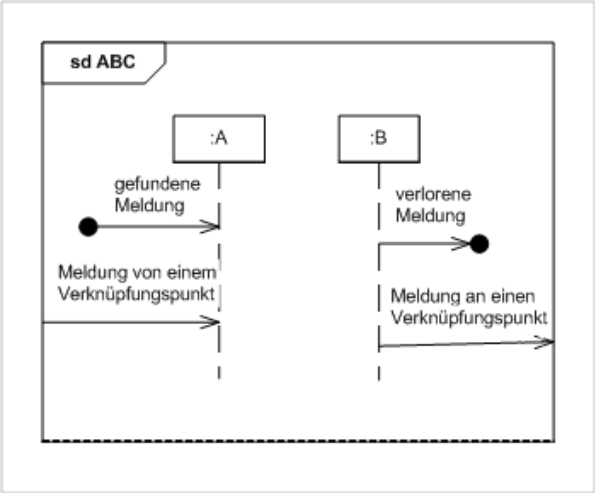
Beispiel eines Sequenzdiagramms mit einem Kopf- und einem Inhaltsbereich

Die Abbildung links zeigt ein Beispiel eines Sequenzdiagramms mit einem Kopf- und einem Inhaltsbereich. Das Schlüsselwort im Kopfbereich ist bei einem Sequenzdiagramm **sd** oder **interaction**. Von jedem Kommunikationspartner geht eine Lebenslinie (gestrichelt) aus. Es sind zwei synchrone Operationsaufrufe, erkennbar an den Pfeilen mit ausgefüllter Pfeilspitze, dargestellt.

Eine Nachricht wird in einem Sequenzdiagramm durch einen Pfeil dargestellt, wobei der Name der Nachricht über den Pfeil geschrieben wird. Synchrone Nachrichten werden mit einer gefüllten Pfeilspitze, asynchrone Nachrichten mit einer offenen Pfeilspitze gezeichnet. Nachrichten, die asynchronen Signalen entsprechen, werden gleich dargestellt wie asynchrone Operationsaufrufe. Die schmalen Rechtecke, die auf den Lebenslinien liegen, sind *Aktivierungsbalken*, die den *Focus of Control* anzeigen, also jenen Bereich, in dem ein Objekt über den Kontrollfluss verfügt, und aktiv an Interaktionen beteiligt ist.



Notationsvarianten für synchrone und asynchrone Nachrichten

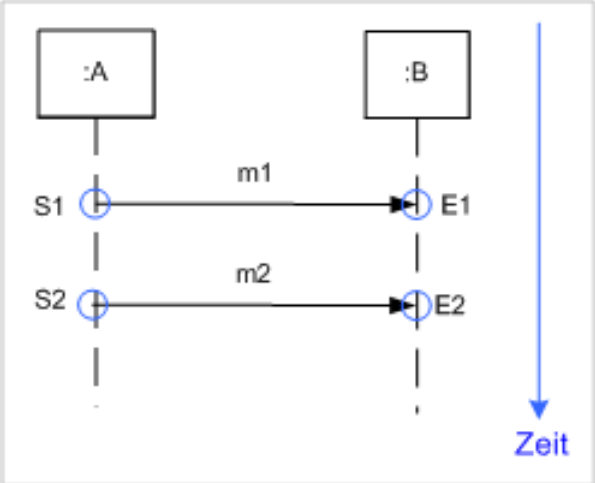


Weitere Notationsvarianten für Nachrichten

Die Abbildung links zeigt vier weitere Notationsvarianten für verlorene und gefundene Nachrichten, sowie für Nachrichten von und an einen Verknüpfungspunkt. Dass es sich um eine Nachricht von einem oder an einen Verknüpfungspunkt handelt, erkennt man daran, dass der entsprechende Pfeil auf dem Rand des Sequenzdiagramms beginnt bzw. endet. Der Verknüpfungspunkt ist einfach der Schnittpunkt des Pfeils mit dem Rand, ein deutlicheres graphisches Symbol ist dafür nicht vorgesehen.

Zeitliche Ordnung der Ereignisse

Ein Sequenzdiagramm beschreibt das Verhalten eines Systems, indem es die zeitliche Ordnung von Ereignisauftritten spezifiziert. Nicht der präzise Zeitpunkt, wann ein Ereignis auftritt, ist dabei ausschlaggebend, sondern welche Ereignisse vor und welche nach einem bestimmten Ereignisauftritt auftreten müssen (Siehe dazu Sequentialisierung und Nebenläufigkeit).



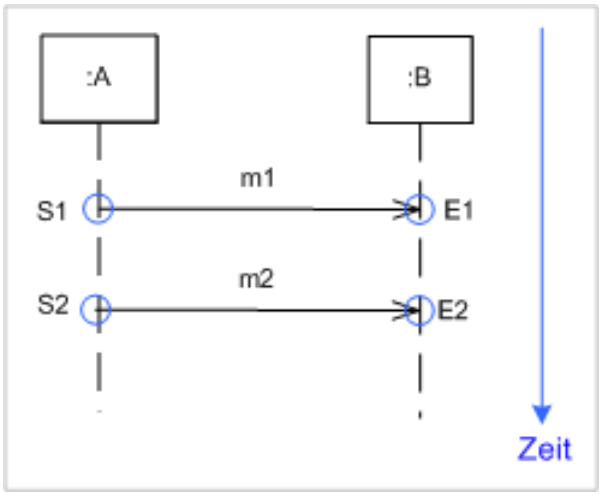
Sequenzdiagramm mit zwei synchronen Operationsaufrufen, das vier Ereignisauftritte modelliert

Die Abbildung links zeigt ein Sequenzdiagramm mit zwei synchronen Operationsaufrufen. Blau eingekreist sind die vier Ereignisauftritte. S1 und E1 stehen für das Sende- und das Empfangs-Nachricht-Ereignis für die Nachricht m1, S2 und E2 für die entsprechenden Ereignisse, die mit m2 in Verbindung stehen. Die Zeitachse läuft in einem Sequenzdiagramm von oben nach unten, sollte aber nicht als absolute Zeit verstanden werden.

Zu den Ereignisauftritten in diesem Sequenzdiagramm lassen sich folgende Aussagen machen:

- E1 tritt nach S1 auf, weil das Empfangs- immer nach dem Sende-Nachricht-Ereignis vorkommt. Analog tritt E2 nach S2 auf.
- S2 tritt nach S1 auf, weil S2 unter S1 gezeichnet ist

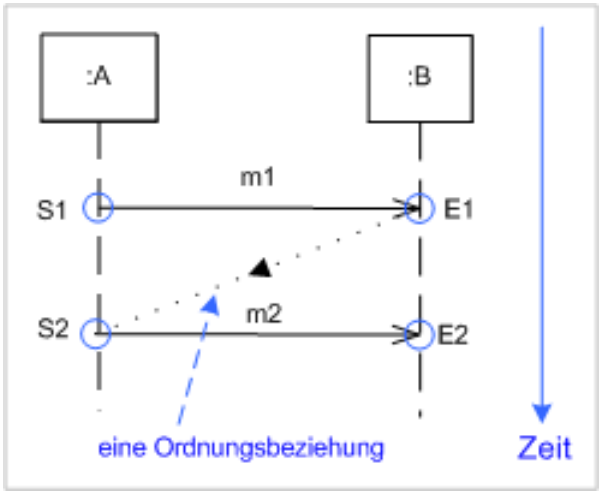
Alles in allem modelliert dieses Sequenzdiagramm also eine Interaktion, die durch genau eine Folge von Ereignisauftritten spezifiziert ist: <S1, E1, S2, E2>.



Sequenzdiagramm mit zwei asychronen Operationsaufrufen, das drei Ereignisauftritte modelliert

Das Sequenzdiagramm in der Abbildung links unterscheidet sich nur geringfügig vom vorangehenden Sequenzdiagramm. Der einzige Unterschied besteht darin, dass statt zwei *synchronen* zwei *asynchrone* Nachrichten dargestellt sind. Hier gilt weiterhin, dass E1 nach S1 und E2 nach S2 auftritt, weil das Empfangs- immer nach dem Sende-Nachricht-Ereignis vorkommt. Weil es sich um asynchrone Kommunikation handelt, könnte E1 hier jedoch nicht nur vor sondern auch erst nach S2 oder E2 vorkommen.

Das Sequenzdiagramm spezifiziert also eine Interaktion, in der drei Spuren von Ereignissauftritten zulässig sind: <S1, E1, S2, E2>, <S1, S2, E1, E2> und <S1, S2, E2, E1>.



Sequenzdiagramm mit zwei asychronen Operationsaufrufen und einer *Ordnungsbeziehung*

Falls nötig, kann man die zulässigen Abfolgen von Ereignisauftritten mit zusätzlichen *Ordnungsbeziehungen* einschränken. Eine Ordnungsbeziehung spezifiziert nicht eine Nachricht, die zwischen zwei Lebenslinien ausgetauscht wird, sondern die Tatsache, dass ein Ereignisauftritt nach einem anderen Ereignisauftritt vorkommen muss. Im Beispiel modelliert die Ordnungsbeziehung, dass S2 immer nach E1 erfolgt.

Mit dieser zusätzlichen Einschränkung stellt dieses Sequenzdiagramm erneut eine Interaktion mit genau einer zulässigen Spur dar: <S1, E1, S2, E2>.

Kombinierte Fragmente

Interaktionen können je nach modelliertem System sehr komplex werden. Wenn es keine Möglichkeit gäbe, Sequenzdiagramme zu modularisieren, wären die entsprechenden graphischen Darstellungen unübersichtlich und schwer verständlich.

Die UML2 hat deshalb aus den Message Sequence Chart deren Konzept der *inline expressions* unter dem Namen *kombinierte Fragmente* übernommen. Ein kombiniertes Fragment ist die Kombination eines *Interaktionsoperators* und eines oder mehrerer *Interaktionsoperanden*. Der Interaktionsoperator spezifiziert die Art des kombinierten Fragments, während die Interaktionsoperanden für die Interaktionsfragmente in diesem kombinierten Fragment stehen.

Ein *Optionales Fragment* besteht zum Beispiel aus dem Interaktionsoperanden **opt**, einer Bedingung und einem Interaktionsfragment. Ist ein optionales Fragment in eine Interaktion eingebunden, wird das zugehörige Interaktionsfragment nur durchlaufen, wenn die Bedingung wahr ist.

Tabelle der kombinierten Fragmente (Lit. : Jeckle 2004, Kapitel 12)

Schlüsselwort	Deutsche Bezeichnung	Englische Bezeichnung	Einsatzzweck
alt	Alternatives Fragment	Alternative	Modellierung alternative Ablaufmöglichkeiten
assert	Zusicherung	Assertion	Modellierung unabdingbarere Interaktionen
break	Abbruchfragment	Break	Modellierung von Ausnahmefällen
consider	Relevante Nachrichten	Consider	Modellierung von Filtern für wichtige Nachrichten
critical	Kritischer Bereich	Critical Region	Modellierung von nicht unterbrechbaren Interaktionen
ignore	Irrelevante Nachrichten	Ignore	Modellierung von Filtern für unwichtige Nachrichten
loop	Schleife	Loop	Modellierung von Iterationen in Interaktionen
neg	Negation	Negative	Modellierung von ungültigen Interaktionen
opt	Optionales Fragment	Option	Modellierung von optionalen Teilen einer Interaktion
par	Paralleles Fragment	Parallel	Modellierung von nebenläufigen Teilen einer Interaktion
seq	Lose Ordnung	Weak Sequencing	Modellierung von Abläufen, die von Lebenslinien und Operanden abhängen
strict	Strenge Ordnung	Strict Sequencing	Modellierung von Abläufen, die *nicht* von Lebenslinien und Operanden abhängen

alt

Durch einen alt-Operator können alternative Abläufe, die durch Bedingungen versehen sind, zusammengefasst werden.

assert

Für eine Nachrichtenmenge kann mit Hilfe dieses Operators eine zwingend notwendige Ablaufreihenfolge angegeben werden.

break

Der normale Ablauf wird unterbrochen, falls eine vorherige Bedingung erfüllt, bzw. verletzt wurde.

consider

Mit Hilfe dieses Operators werden nur die angegebenen Aktionen ausgeführt, der Rest wird ignoriert.

critical

Falls diese Region betreten wird, so werden alle Aktionen ohne jegliche Unterbrechung ausgeführt.

ignore

Bestimmte Aktionen können mit Hilfe dieses Operators an der Ausführung gehindert werden.

loop

Mit Hilfe des loop-Operators können Schleifen definiert werden. Zur Vereinfachung findet man manchmal auch loop while oder loop until.

neg

Dieser Operator kapselt unzulässige Abläufe.

opt

Die einfachste Form der Operatoren ist der opt-Operator, der optionale Teilabläufe umfasst.

par

Der par-Operator dient der Darstellung von parallelen Abläufen

ref

Mit Hilfe dieses Operators wird durch eine Referenz auf ein anderes Sequenzdiagramm verwiesen, das einen Teilablauf beschreibt.

seq

Legt eine Reihenfolge für die Abfolge von Aktionen einer Lebenslinie vor.

strict

Ähnelt dem Aufbau des seq-Operators. Hier jedoch betrifft die Reihenfolge nicht nur eine Lebenslinie, sondern gleich alle Lebenslinien.

Siehe auch

- Interprozesskommunikation
- Prozessynchronisation
- UML (Unified Modeling Language)

Weblinks

- Ein einfacher Editor für Sequenzdiagramme [1] (Java)
- Ein einfacher Online-Editor für Sequenzdiagramme [2]

Literatur

- Christoph Kecher: *UML 2.0 - Das umfassende Handbuch.* 2. Auflage. Galileo Press, Bonn 2006, ISBN 978-3-89842-738-8.
- Heide Balzert: *Lehrbuch der Objektmodellierung - Analyse und Entwurf mit der UML 2.* 2. Auflage. Elsevier Spektrum Akademischer Verlag, Heidelberg, München 2005, ISBN 3-8274-1162-9.
- M. Jeckle , C. Rupp, J. Hahn, B. Zengler, S. Queins: *UML 2 glasklar.* Hanser, München, Wien 2004, ISBN 3-446-22575-7, Kapitel 12 - Sequenzdiagramm.
- ITU-T Recommendation (Hrsg.): *Message Sequence Chart (MSC).* In: *Languages and general Software Aspercts for Telecommunication Systems.* Nr. Z.120, November 1999 (PDF [3], abgerufen am 30. April 2009).

Referenzen

[1] http://sourceforge.net/projects/sdedit
[2] http://www.websequencediagrams.com
[3] http://www.itu.int/ITU-T/studygroups/com10/languages/Z.120_1199.pdf

Anwendungsfalldiagramm

Strukturdiagramme der UML
Klassendiagramm
Komponentendiagramm
Kompositionsstrukturdiagramm
Objektdiagramm
Paketdiagramm
Verteilungsdiagramm
Profildiagramm
Verhaltensdiagramme der UML
Aktivitätsdiagramm
Anwendungsfalldiagramm
Interaktionsübersichtsdiagramm
Kommunikationsdiagramm
Sequenzdiagramm
Zeitverlaufsdiagramm
Zustandsdiagramm

Ein **Anwendungsfalldiagramm** (engl. *use case diagram*) ist eine der 14 Diagrammarten der Unified Modeling Language (UML), einer Sprache für die Modellierung der Strukturen und des Verhaltens von Software- und anderen Systemen. Es stellt Anwendungsfälle und Akteure mit Ihren jeweiligen Beziehungen dar.

Das Anwendungsfalldiagramm ist ein *Verhaltensdiagramm*. Es zeigt eine bestimmte Sicht auf das *erwartete* Verhalten eines Systems und wird deshalb für die Spezifikation der Anforderungen an ein System eingesetzt. In einem Anwendungsfalldiagramm werden typischerweise Anwendungsfälle und Akteure mit ihren Abhängigkeiten und Beziehungen dargestellt.

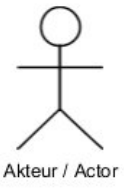
Ein Anwendungsfalldiagramm stellt keine Ablaufbeschreibung dar. Ablaufbeschreibungen zu einem Anwendungsfall können mit einem Aktivitäts-, einem Sequenz- oder einem Kollaborationsdiagramm / (ab UML 2.x Kommunikationsdiagramm) dargestellt werden.

Anwendungsfalldiagramm in Stichpunkten

- **Ziel** ist es, möglichst einfach zu zeigen, was man mit dem zu bauenden Softwaresystem machen will. Welche Fälle der Anwendung es also gibt.
- **Akteure** werden als „Strichmännchen“ dargestellt, welche sowohl Personen wie Kunden oder Administratoren als auch ein System darstellen können.
- **Anwendungsfälle** werden in Ellipsen dargestellt. Sie müssen beschrieben werden (z.B. in einem Kommentar oder einer eigenen Datei).
- **Assoziationen** zwischen Akteuren und Anwendungsfällen müssen durch Linien gekennzeichnet werden.
- **Systemgrenzen** werden durch Rechtecke gekennzeichnet.
- **include-Beziehungen** werden mittels (mit <<*include*>> gekennzeichneter) gestrichelter Linie und einem Pfeil zum inkludierten Anwendungsfall gekennzeichnet, wobei dieser für den aufrufenden Anwendungsfall notwendig ist.
- **extend-Beziehungen** werden mittels (mit <<*extend*>> gekennzeichneter) gestrichelter Linie und einem Pfeil vom erweiternden Anwendungsfall gekennzeichnet, wobei dieser von dem aufrufenden Anwendungsfall aktiviert werden kann, aber nicht muss.

Elemente

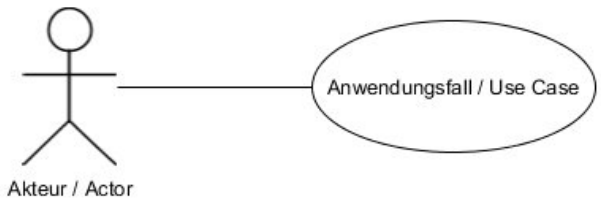
Der **Systemkontext** wird durch **Systemgrenzen** in Form von Rechtecken gekennzeichnet.



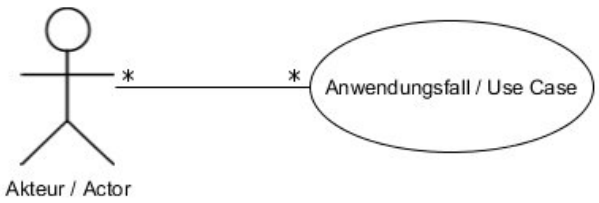
Akteure werden als „Strichmännchen“ dargestellt, welche sowohl Personen wie Kunden oder Administratoren als auch ein System darstellen können.

Anwendungsfälle werden in Ellipsen dargestellt. Sie müssen beschrieben werden (z.B. in einem Kommentar oder einer eigenen Datei).

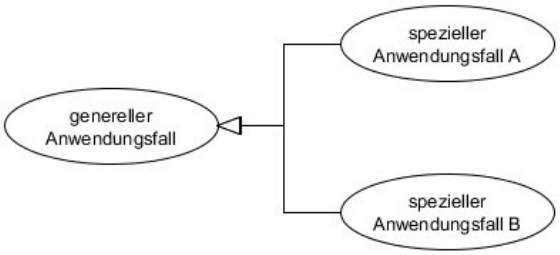
Beziehungen



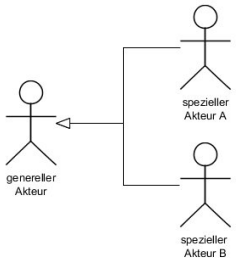
Assoziation / Kommunikation von Akteur und Anwendungsfall.



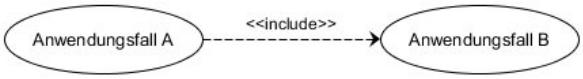
Multiplizität von Akteur und Anwendungsfall wobei die Voreinstellung des Akteurs 1 ist.



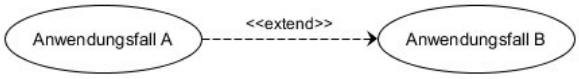
Generalisierung von Anwendungsfällen.



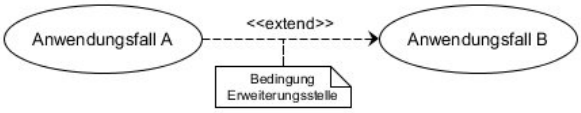
Generalisierung von Akteuren.



Include-Beziehung im Anwendungsfalldiagramm wobei Anwendungsfall A den Anwendungsfall B beinhaltet.



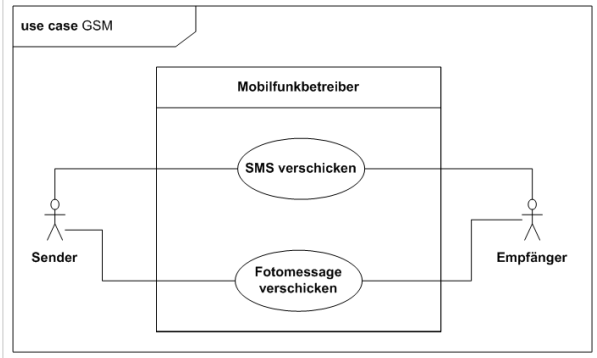
Extend-Beziehung im Anwendungsfalldiagramm wobei Anwendungsfall A den Anwendungsfall B erweitert.



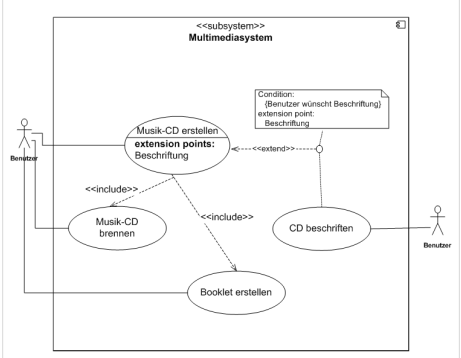
Extend-Beziehung mit extension point wobei Anwendungsfall A den Anwendungsfall B unter der angegebenen Bedingung erweitert.

Anwendungsfall mit extension point.

Beispiele



Das Schlüsselwort im Kopfbereich ist *use case*. Das Anwendungsfalldiagramm in der Abbildung links ist in einen Kopf- und in einen Inhaltsbereich getrennt und mit einem Rahmen umschlossen, so wie es die UML2 neu für alle Diagramme vorsieht.



Die Abbildung rechts zeigt ein komplexeres Anwendungsfalldiagramm, das die Beziehungen zwischen dem Akteur Benutzer und dem System Multimediasystem festhält. Ein Benutzer ist an vier Anwendungsfällen interessiert, die ihrerseits untereinander in Beziehung stehen. Musik-CD erstellen ist der komplexeste Anwendungsfall, weil er zwei andere Anwendungsfälle importiert und optional durch einen dritten, CD beschriften, erweitert wird.

Unterschiede zur UML 1.x

Das Anwendungsfalldiagramm wird in der UML2 neu als *Verhaltensdiagramm* und nicht mehr als Strukturdiagramm eingestuft. Des Weiteren müssen Akteure nun einen Namen haben und die Vorbedingungen der jeweiligen extension point müssen per Notiz an die entsprechende Erweiterungsbeziehung angehängt werden.

Siehe auch

- Kontextdiagramm

Weblinks

- Notationsübersicht UML 2.0 [1] (PDF-Datei; 304 kB)
- UML 2 glasklar - ausführliche Leseprobe [2] (PDF; 691 kB)

Literatur

- Christoph Kecher: "UML 2.0 - Das umfassende Handbuch" Galileo Computing, 2006, ISBN 3-89842-738-2
- Bernd Oestereich: "Analyse und Design mit UML 2" Oldenbourg Wissenschaftsverlag, 2006, ISBN 3-486-57926-6

Referenzen

[1] http://www.oose.de/downloads/uml-2-Notationsuebersicht-oose.de.pdf

[2] http://files.hanser.de/hanser/docs/20070802_2782114221-80_978-3-446-41118-0_Leseprobe.pdf

Grafische Benutzeroberfläche

Eine **grafische Benutzeroberfläche** ist eine Software-Komponente, die dem Benutzer eines Computers die Interaktion mit der Maschine über grafische Symbole erlaubt. Die Darstellungen und Elemente (Arbeitsplatz, Symbole, Papierkorb, Menü) können meist unter Verwendung eines Zeigegerätes wie einer Maus gesteuert werden.

Synonyme Bezeichnungen sind die Abkürzung **GUI** (engl. „**G**raphical **U**ser **I**nterface“) und dessen wörtliche Übersetzung **grafische Benutzerschnittstelle**. Im Gebiet der Software-Ergonomie werden stattdessen die Begriffe „grafische Benutzungsschnittstelle“ oder „Mensch-Maschine-Schnittstelle“ verwendet. In der Breite haben GUIs die auf Zeichen basierenden Benutzerschnittstellen CLI (command line interface) abgelöst.

KDE 4.3 – eine Benutzeroberfläche, die meist auf Unix-Betriebssystemen Verwendung findet

Geschichte

Das Konzept von GUIs im heutigen Sinne stammt aus den 1970er Jahren. Seit 1973 erarbeitete man am Xerox PARC in Kalifornien den Xerox Alto. Den ersten kommerziellen Einsatz zeigte 1981 der Xerox Star. Einen größeren Kreis von Anwendern erreichte das Konzept erst durch die populäreren Computer von Apple. Ab 1979 wurde dort daran gearbeitet, wobei man sich sehr von Xerox inspirieren ließ, und 1983 erschien der Apple Lisa mit grafischer Benutzeroberfläche. Dieser war noch überaus teuer, wichtiger war für die Zukunft der Apple Macintosh von 1984, der unter der Leitung von Steve Jobs entwickelt wurde.

Microsoft kam im November 1985 mit Windows (1.03) hinzu, das bereits zwei Jahre zuvor angekündigt worden war (als Reaktion auf Lisa) und die hohen Erwartungen kaum erfüllte.[1] Es lief auf den IBM-kompatiblen PCs und setzte sich später gegen das von IBM bevorzugte OS/2 durch. Auch für den weit verbreiteten Commodore 64 erschien eine grafische Benutzeroberfläche, GEOS von 1986. Der Atari ST und der Amiga (ebenfalls von Commodore) waren bereits als GUI-Computer konzipiert worden; sie galten als Semi-PCs, die billiger als PCs, aber leistungsfähiger als Heimcomputer wie der Commodore 64 waren.

Dennoch setzten sich GUIs nur langsam durch, da die damaligen Computer meist noch zu langsam waren, um das Konzept angemessen zu realisieren. Als die Zeitschrift *64'er* im Mai-Heft 1990 vier Benutzeroberflächen miteinander verglich, erhielten der Commodore 64 (mit GEOS) und der AT 286 (also ein IBM-PC, mit Windows) die Note gut, der Amiga und der Atari nur ein befriedigend. Die Zeitschrift lobte die größere Benutzerfreundlichkeit von GUIs, wies aber auf das Problem hin, dass für manche nur wenige Anwendungen existieren. Als einheitliche Lösung mit großem Umfeld an Anwendungen fiel der Windows-PC positiv auf, er war aber auch am teuersten: Gerät (mit Diskettenlaufwerk, Monitor und Maus) und Software kosteten damals 4000 DM, das entsprechende Paket Commodore 64/GEOS bzw. der Amiga waren nur halb so teuer. Der Atari ST kostete nur 1200 DM, arbeitete aber nur schwarz-weiß und wurde mit wenig Software ausgeliefert.[2]

8-Bit-Rechner wie der Commodore 64 erwiesen sich letztlich als zu langsam; wegen des begrenzten Arbeitsspeichers musste man des öfteren mit Disketten hantieren. Daher waren grafische Benutzeroberflächen eher erst für die Generation der 16-Bit-Rechner geeignet, zum Beispiel für den Atari ST. Der Durchbruch von Microsoft Windows erfolgte nach 1992 mit Windows 3.1. Windows gilt heute als Standard beim Arbeiten mit PCs.

Unter Unix und Linux gibt es mehrere, auf das X Window System aufsetzende Desktop-Umgebungen, die den Zweck einer grafischen Oberfläche erfüllen. Besonders bekannte Vertreter sind die Desktop-Umgebungen KDE sowie des Weiteren GNOME, Xfce und Enlightenment, ferner die Lightweight-Entwicklung LXDE.

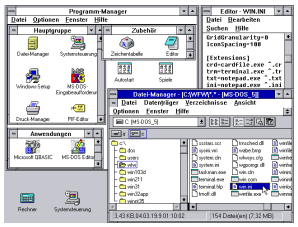
Mit zunehmendem Funktionsumfang der GUIs selbst und der zugehörigen Programme nahm auch der Ressourcenbedarf der betreffenden Betriebssysteme immer weiter zu. Reichten z. B. für GEOS 1.2 noch ein 8-Bit-Prozessor mit 1 MHz, 64 KB Arbeitsspeicher und ein 5¼-Zoll-Diskettenlaufwerk mit 170-KB-Disketten, so empfahl Microsoft für die Version von Windows Vista Home Basic (Oktober 2007) einen 32-Bit-Prozessor mit 800 MHz, 512 MB Arbeitsspeicher, eine 20-GB-Festplatte (davon 15 GB frei) und ein DVD-ROM-Laufwerk.[3]

Windows 1.03 (1985)

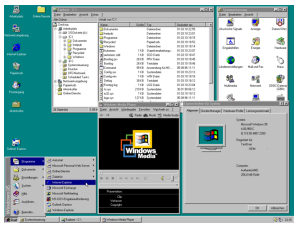
GEOS auf einem Commodore 64 (1986)

TOS 1.00 auf einem Atari 1040 ST (1986)

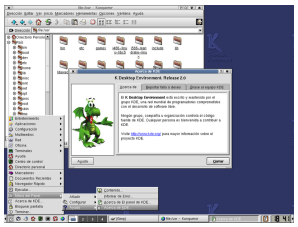
Amiga 500 (1987)



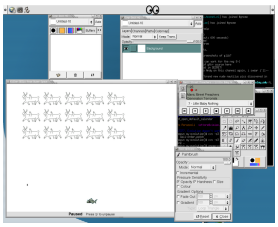
Windows 3.1 (1992)



Windows 95 (1995)



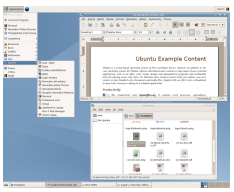
KDE 2.0 / Mandrake (2000)



GNOME (2002)

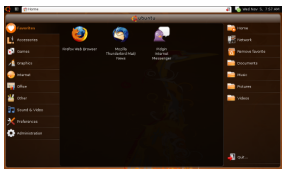
Mac OS X auf einem iBook G4 von Apple (2003)

GNOME / Ubuntu (2005)



Xfce / Xubuntu (2006)

KDE 3.5.9 mit Fenstermanager Compiz(Beryl) / PCLinuxOS (2007)



"Launcher", Ubuntu Notebook Remix (2009)

Windows 7 GUI (2009)

Normierung der Anforderungen

Die Anforderungen an eine grafische Benutzungsschnittstelle im Rahmen der Mensch-Computer-Kommunikation sind in der europäischen Norm EN ISO 9241-10 ff. geregelt. Dabei muss die Schnittstelle folgende Merkmale aufweisen:

- Aufgabenangemessenheit
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Erwartungskonformität
- Fehlertoleranz
- Individualisierbarkeit
- Lernförderlichkeit

Ferner ist in der Norm EN ISO 9241 die Umsetzung von Benutzungsschnittstellen für Web-Applikationen und deren Evaluation im Rahmen der Benutzbarkeit definiert.

GUI-Elemente

Ein GUI hat die Aufgabe, Anwendungssoftware auf einem Rechner mittels grafischer Elemente, Steuerelemente oder auch Widgets genannt, bedienbar zu machen. Dies geschieht meistens mittels einer Maus als Steuergerät, mit der die grafischen Elemente bedient oder ausgewählt werden. Die Gesamtgestaltung heutiger grafischer Oberflächen verwendet oftmals die sogenannte Desktop-Metapher.

Programme öffnen dabei zunächst ein Hauptfenster. Das GUI-System erlaubt, solche Fenster in ihrer Größe und Position zu verändern, auszublenden oder auf die gesamte Bildschirmgröße zu vergrößern. Grafische Bedienoberflächen sind für viele Mehrzweck-Betriebssysteme verfügbar oder gar in sie integriert. Weitere Bedienelemente sind Schaltflächen (Buttons), Toolbars (Werkzeugleisten), Schieberegler, Auswahllisten und Symbole. Darüber hinaus werden Dialogboxen (auch „Dialogfelder“ genannt) meist für Benutzerabfragen oder Eingaben verwendet wie beispielsweise die Auswahl eines Druckers.

GUIs können mit der Verwendung von Metaphern für bestimmte Programmfunktionen, wie zum Beispiel dem Papierkorb, das Erlernen und das Verständnis der Bedienung wesentlich erleichtern.

Kein separates GUI-Element, aber relevant für alle GUI-Elemente ist der Fokus: Das GUI-Element, welches aktuell für die nächste Benutzer-Aktion (Eingabe von Daten, Änderungs des Zustands etc.) relevant ist, besitzt den Fokus.

Damit der Fokus jederzeit für den Benutzer sichtbar ist, ist er grafisch hervorgehoben: in textuellen Eingabefeldern durch eine blinkende Eingabemarke (Cursor, Caret); andere GUI-Elemente sind meist durch eine gepunktete, dünne Umrandung hervorgehoben, wenn sie fokussiert sind (den Fokus besitzen).

Siehe auch

- Liste von GUI-Bibliotheken
- Smalltalk
- Benutzerfreundlichkeit
- Skalierbare Benutzeroberfläche
- WIMP (Benutzerschnittstelle)
- Benutzerschnittstelle

Weblinks

- Software-Ergonomie-Glossar [4]
- Leitlinien für die Gestaltung von ergonomischen WWW-Informationssystemen. Alte Informationen von 1997–2002 [5]

Einzelnachweise

[1] Daniel Ichbiah: *Die Microsoft Story. Bill Gates und das erfolgreichste Software-Unternehmen der Welt*, Heyne: München 1993 (Original 1991), S. 241, S. 253–256.
[2] Dirk Astrath: Ganz und gar nicht oberflächlich. In: *64'er*, Mai 1990, S. 54–60.
[3] Windows Vista Home Basic (PDF-Datei) (http://download.microsoft.com/download/7/5/1/751848cf-328c-480b-8477-6a333bc74da0/Windows_Vista_Home_Basic.pdf)
[4] http://www1.informatik.uni-jena.de/Lehre/SoftErg/vor_glos.htm
[5] http://vsis-www.informatik.uni-hamburg.de/ergonomie/

Systementwurf

Chipentwurf (oder *Chipentwicklung*) bezeichnet in der Mikroelektronik den Prozess der Entwicklung eines Mikrochips von der ersten Idee über die Spezifikation und Umsetzung in einen Schaltplan und ein Layout bis zum gefertigten Schaltkreis.

Entwurfsmethoden

Die Probleme aufgrund zunehmender Komplexität sowie der Kosten- und Zeitdruck haben zur Entwicklung unterschiedlicher Methodiken des Chipentwurfs geführt. Allen Methoden gemein ist, dass Masken für die Fotolithografie verwendet werden, mit denen eine Fertigung in der Halbleitertechnik möglich ist. Die Unterschiede der Methoden bestehen im Entwicklungsaufwand und in der Entwurfsflexibilität.

Die Bezeichnung der Methodik hängt oftmals eng mit dem gewünschten Produkt zusammen und lässt sich folgendermaßen einordnen:

- Entwurf als Standardschaltung
  - Fest verdrahteter Vollentwurf. Dieses ist die klassische Methodik, die alle Möglichkeiten der Halbleitertechnik bietet, inklusive analoger Schaltungen. Hiermit werden vor allem Massenprodukte hergestellt, wie z. B. Mikroprozessoren, RAM-Bausteine und Produkte mit besonderen Anforderungen z. B. Analog-digital-Umsetzer, Bausteine der Automobilelektronik.
  - Maskenprogrammierung. Die Funktion ist mit einer Entwurfsmethodik wie oben festgelegt, jedoch können die Inhalte für integrierte Speicher durch Änderung der Fotomasken nur des letzten Fertigungsschrittes festgelegt (verdrahtet) werden. Beispiele: ROM, Mikrocontroller.
  - Anwendungsspezifische Programmierung. Auf Basis eines Standardbauteils können darin vorhandene logische Grundelemente durch nachträgliche Programmierung verbunden werden, ohne dass erneut lithografische Masken benötigt werden. Beispiele: PROM, PLD, PLA, FPGA.
- Entwurf als anwenderspezifische Schaltung (ASIC)
  - Anwendungsspezifischer Entwurf. Wird für hochspezialisierte Bausteine in meist kleineren Stückzahlen verwendet.
    - Full-Custom-Entwurf
    - Semi-Custom-Entwurf
      - Entwurf mit Standardzellen
      - Gate Array / Sea-of-Gates

- FPGA-Entwurf. Ähnlich wie oben, jedoch wird Funktion beim Programmieren fest eingebrannt, z. B. mit Antifuse-Technik

Full-Custom-Entwurf

Die Entwicklung des Chips oder der integrierten Schaltung erfolgt nicht mit vorgefertigten Zellen oder Schaltungsteilen, sondern individuell an die Anforderungen der zu entwerfenden Schaltung angepasst. Grundsätzlich unterscheidet man analoge und digitale Schaltungen. Im Bereich der analogen Schaltungstechnik verwendet man fast ausschließlich den Full-Custom-Entwurf. Man hat die Möglichkeit, jeden einzelnen Transistor so zu verschalten und zu dimensionieren, wie es für die Funktion der Schaltung nötig ist. Im Bereich der digitalen Schaltungstechnik wird häufig der Semi-Custom-Entwurf verwendet. Dieser schränkt jedoch den Entwurf meist stark ein, da dabei im wesentlichen statische Logik Verwendung findet. Möchte man hingegen andere Logiktechniken wie beispielsweise dynamische Logik verwenden, so greift man auf den flexibleren Full-Custom-Entwurf zurück.

Der Full-Custom-Entwurf ist wesentlich zeitaufwendiger, da er nicht so stark automatisiert ist wie der Semi-Custom-Entwurf. Er bietet jedoch die Möglichkeit, die Schaltung bezüglich der Leistungsaufnahme, der Geschwindigkeit und der benötigten Chipfläche zu optimieren. Durch die großen Freiheiten beim Entwurf sowie bei der Ausführung des Layouts ist enormes Optimierungspotential vorhanden.

Für den Full-Custom-Entwurf werden dem Entwickler vom Halbleiterhersteller sogenannte Design Kits zur Verfügung gestellt. Dies sind Software-Bibliotheken für das jeweils verwendete Design-Tool, welche dem Entwickler eine Anzahl primitiver Bauteile (Transistoren, Widerstände etc.) mit den dazugehörigen, aus Messdaten gefertigter Schaltungen gewonnenen, Softwaremodellen für die Schaltungssimulation bieten.

Das Layout eines Full-Custom-Entwurfs wird manuell generiert. Dabei kann der Entwickler die Geometrien der einzelnen Transistoren und der Metalleitungen bestimmen und optimieren. Dabei müssen Einschränkungen der Fertigung in Form von geometrischen und elektrischen Regeln (Design Rules) eingehalten werden.

Semi-Custom-Entwurf

Beim *Semi-Custom-Entwurf* sind die Freiheiten des Entwicklers weiter eingeschränkt. Dadurch wird der Entwicklungsprozess jedoch einfacher, da vermehrt auf vorgefertigte Elemente zurückgegriffen wird.

Etwas in der Bedeutung verloren haben die *Gate-Arrays* oder *Sea-Of-Gates*. Bei beiden handelt es sich um halbfertige Bausteine, bei denen die Transistoren bereits platziert sind. Die logischen Elemente entstehen durch Festlegung der Verdrahtungsebenen (Metallagen) nur mit den dafür zuständigen Fotomasken. Dadurch können prinzipiell Kosten gespart werden. Der Entwurfsprozess ist jedoch eingeschränkt durch begrenzten Raum für Verdrahtung. Insbesondere bei den Gate-Arrays sind im Gegensatz zu Sea-of-Gates nur bestimmte Gebiete (Verdrahtungskanäle) für Verbindungen zugelassen. Weiterhin sind die Stärken der Gatter nicht variabel genug. Die daraus resultierenden Nachteile sind: Hoher Stromverbrauch sowie geringe Funktionsdichte und daraus resultierend hohe Stückkosten.

Verbreitet ist der Entwurf mit *Standardzellen*. Standardzellen sind vorentworfene Elemente vom einfachen Gatter über Flip-Flops bis hin zu RAM oder Prozessoren. Auch analoge Blöcke wie Analog-digital-Umsetzer sind möglich. Die Zellen können frei im Layout platziert werden, haben aber bekannte elektrische und geometrische Parameter. Diese Parameter sind in sog. Bibliotheken abgelegt und werden von den Entwicklungswerkzeugen abgerufen. Im Layout wird die Schaltung durch grafisches Aneinanderreihen und Verbinden der Standardzellen erzeugt. Dadurch wird der Entwicklungsprozess gegenüber dem Full-Custom-Entwurf deutlich einfacher, weil ein Großteil der Schaltungssimulation auf logischer Ebene gemacht werden kann. Bei hohen Stückzahlen (> 100.000) ist der Semi-Custom-Entwurf der beste Kompromiss zwischen der Effektivität des Chipentwurfs und Kosten/Qualität des resultierenden Bausteins.

Bei geringeren Stückzahlen und komplexen Funktionen bieten sich FPGAs an. Die Entwurfsmethodik hat sich mit steigender Komplexität immer mehr der des Semi-Custom-Entwurfs angenähert. Im Gegensatz dazu sind die

logischen Elemente beim FPGA bereits auf dem Chip vorhanden und werden lediglich durch vorübergehendes oder dauerhaftes Programmieren (*Brennen*) verbunden. Verwendet werden vorproduzierte integrierte Schaltkreise, die als Standardschaltung entworfen wurden. Eine wesentliche Ersparnis an Zeit und Kosten resultiert daher, dass die entworfene Funktion "im Feld", d.h. in wenigen Minuten beim Anwender auf den Baustein aufgebracht werden kann. Nachteilig sind jedoch die teilweise sehr hohen Kosten, großen Bauformen und der Stromverbrauch dieser Bausteine.

Entwurfsprozess („Designflow“)

Alle komplexen digitalen integrierten Schaltungen werden grob nach folgendem Schema entwickelt, das sich stark auf Werkzeuge zur Entwurfsautomatisierung abstützt:

1. Spezifikation (Festlegung der Funktion in Worten und Bildern)
2. Beschreibung und Validierung auf Verhaltensebene (Eingabe und Simulation beispielsweise in C/C++ oder MATLAB)
3. Beschreibung und Validierung auf RTL-Ebene (Eingabe und Simulation in einer Hardwarebeschreibungssprache, siehe unten)
4. Synthese (Erzeugung von Gattern aus der RTL-Beschreibung)
5. Validierung auf Gatterebene (Simulation oder formale Verifikation)

Kern des modernen Entwurfsprozess ist die Beschreibung der Funktion auf einer höheren Abstraktionsebene, die *RTL (Register Transfer Level)* genannt wird. Hier können komplexe Funktionen in einer Hardwarebeschreibungssprache (ähnlich einer Programmiersprache) (z. B. Addition, Multiplikation) eingegeben und die Gesamtfunktion der eingegebenen RTL-Beschreibung am Computer simuliert werden. Als Hardwarebeschreibungssprachen kommen meist VHDL oder Verilog zum Einsatz. Eine grafische Eingabe des Schaltplans auf RTL- oder Gatterebene ist möglich, aber für größere Schaltungen meist nicht praktikabel.

Die RTL-Beschreibung wird dann mit einem Synthesewerkzeug, das ähnlich wie ein Compiler arbeitet, in eine Gatterbeschreibung, die sogenannte Netzliste umgesetzt. Diese Netzliste kann zur Kontrolle des Ergebnisses erneut simuliert werden, da sie in der Regel dieselbe VHDL- oder Verilog-Syntax verwendet.

Die Synthese auf Verhaltensebene und die Validierung auf Spezifikationsebene sind im Gegensatz dazu derzeit (2006) im allgemeinen nicht automatisiert.

Zur Vorbereitung der Fertigung sind weiterhin folgende Schritte nötig:

- Floorplanning (grobe Platzierung der hierarchischen Blöcke und Globalverdrahtung)
- Layoutsynthese (Erstellung einer detaillierten geometrischen Anordnung bis zur untersten Ebene)
- Statische Timing Analyse ('STA')
- Layoutverifikation (Überprüfung der elektrischen Designregeln 'ERC' und der geometrischen und sonstigen Designregeln 'DRC') sowie Schaltungsvergleich (im Allgemeinen mit der Gatterebene 'LVS')
- Tape-Out (Abgabe bei der Fertigung)

Bei FPGAs wird das Layout nicht (wie bei Semi- oder Full-Custom-Entwürfen) zur Herstellung von Fotomasken, sondern zur Erzeugung der Programmierung der Verbindungen verwendet. Die Schritte für *Design Rule Check* und *Tape-Out* entfallen damit.

Die Taktfrequenz ist begrenzt durch die Addition der Signallaufzeiten durch die Schaltelemente und der Verdrahtung. Werden moderne VLSI Fertigungsprozesse mit Strukturgrößen von 130 nm (deep submicron) oder kleiner verwendet, steigt der Einfluss der Verbindungen auf die Signallaufzeiten. Dies wirkt sich reduzierend auf die erreichbare Taktfrequenz aus, mit welcher der Baustein korrekt arbeiten kann. Die Ergebnisse des Layouts koppeln also auf die Funktion zurück, was zwar berechenbar ist, aber bei Ziel-Verfehlung dennoch weitere Design-Zyklen nötig macht.

Die Laufzeiten der Gatter und der Verdrahtung werden mittels „statischer Timing-Analyse“ aufaddiert und dargestellt. Dabei wird der *kritische Pfad* ermittelt. Dieser stellt den längst möglichen Weg dar, den ein Signal während einer Taktperiode in der Schaltung nehmen kann und legt damit die maximale Arbeitsfrequenz der gesamten Schaltung fest. Dabei sind Fertigungstoleranzen und Temperatureffekte zu berücksichtigen. Heutige Designs skalieren die Taktraten im System mit zunehmendem Abstand vom Kern herunter oder/und spalten den Halbleiter in einzelne logische Recheneinheiten auf, die dann ihre Operationen nur in einem verhältnismäßig kleinen Bereich aber mit hoher Frequenz und synchron durchführen.

Die fertigen Geometriedaten werden an die Fertigung übergeben, wo sie zur Herstellung der fotografischen Masken verwendet werden. Der Vorgang wird als Tape-Out bezeichnet, weil dafür früher Magnetbänder (tapes) verwendet wurden. Die Einhaltung aller Designregeln ist wichtig, damit die Ausbeute (yield) an funktionsfähigen Bauteilen in der Fabrik möglichst hoch ist.

Literatur

- Jens Lienig: *Layoutsynthese elektronischer Schaltungen - Grundlegende Algorithmen für die Entwurfsautomatisierung.* Springer, Berlin 2006, ISBN 978-3-5402-9627-0.
- Peter Marwedel: *Synthese und Simulation von VLSI- Systemen.* Hanser Fachbuchverlag, 1993, ISBN 978-3-4461-6146-7.
- Franz J. Rammig: *Systematischer Entwurf digitaler Systeme.* Vieweg+Teubner, Stuttgart 1989, ISBN 978-3-5190-2265-7.
- Erich Barke et al.: *Electronic Design Automation – Entwurfsautomatisierung in der Mikroelektronik* [1]. Vorlesungsskript, Universität Hannover.

Weblinks

- Deutsches Zentrum für Entwurfsautomatisierung [2]

Referenzen

[1] http://edascript.ims.uni-hannover.de/

[2] http://www.edacentrum.de/

Unified Modeling Language

Die **Unified Modeling Language**, kurz **UML** (auf deutsch „Vereinheitlichte Modellierungssprache“), ist eine graphische Modellierungssprache zur Spezifikation, Konstruktion und Dokumentation von Teilen von Software und anderen Systemen[1] . Sie wird von der Object Management Group (OMG) entwickelt und ist sowohl von ihr als auch von der ISO (ISO/IEC 19501 für Version 2.1.2[2]) standardisiert. Im Sinne einer Sprache definiert UML dabei Bezeichner für die meisten für die Modellierung wichtigen Begriffe und legt mögliche Beziehungen zwischen diesen Begriffen fest. UML definiert weiter graphische Notationen für diese Begriffe und für Modelle von statischen Strukturen und von dynamischen Abläufen, die man mit diesen Begriffen formulieren kann.

UML ist heute eine der dominierenden Sprachen für die Modellierung von betrieblichen Anwendungs- bzw. Softwaresystemen. Der erste Kontakt zu UML besteht häufig darin, dass Diagramme in UML im Rahmen von Softwareprojekten zu erstellen, zu verstehen oder zu beurteilen sind:

- Projektauftraggeber und Fachvertreter prüfen und bestätigen zum Beispiel Anforderungen an ein System, die Wirtschaftsanalytiker in Anwendungsfalldiagrammen in UML festgehalten haben.
- Softwareentwickler realisieren Arbeitsabläufe, die Wirtschaftsanalytiker in Zusammenarbeit mit Fachvertretern in Aktivitätsdiagrammen beschrieben haben.
- Systemingenieure installieren und betreiben Softwaresysteme basierend auf einem Installationsplan, der als Verteilungsdiagramm vorliegt.

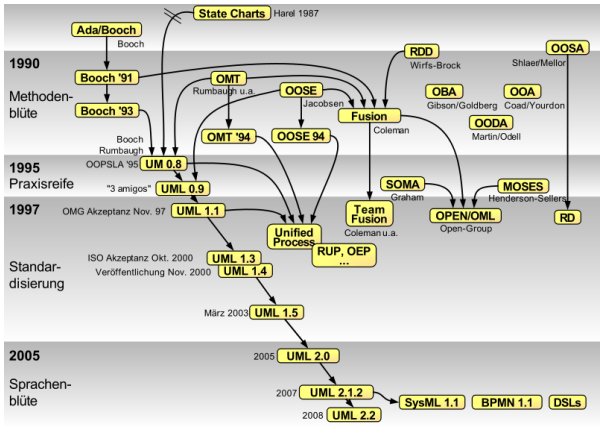
Die graphische Notation ist jedoch nur ein Aspekt, der durch UML geregelt wird. UML legt in erster Linie fest, mit welchen Begriffen und welchen Beziehungen zwischen diesen Begriffen sogenannte *Modelle* spezifiziert werden – Diagramme in UML zeigen nur eine graphische Sicht auf Ausschnitte dieser Modelle. UML schlägt weiter ein Format vor, in dem Modelle und Diagramme zwischen Werkzeugen ausgetauscht werden können.

Entstehungsgeschichte

Die erste Version von UML (»Programmablaufplan Reloaded«) entstand in den 1990er Jahren als Reaktion auf zahlreiche Vorschläge für Modellierungssprachen und -methoden, welche die zu dieser Zeit aufkommende objektorientierte Softwareentwicklung unterstützen sollten. Die erste Folge von Sprachversionen, auch bekannt unter dem Namen UML 1.x, wurde 2005 durch eine grundlegend überarbeitete Version, oft als UML2 bezeichnet, abgelöst.

Von den Anfängen zur Unified Modeling Language 1.x

Die Väter von UML, insbesondere Grady Booch, Ivar Jacobson und James Rumbaugh, auch „die drei Amigos“ genannt, waren in den 1990er-Jahren bekannte Vertreter der objektorientierten Programmierung. Sie hatten alle bereits ihre eigenen Modellierungssprachen entwickelt. Als sie zusammen beim Unternehmen Rational Software beschäftigt waren, entstand die Idee, die verschiedenen Notationssysteme strukturiert zusammenzuführen. Eine Vielzahl von unterschiedlichen Modellierungssprachen hatte direkten oder indirekten Einfluss auf die Konzeption von UML, darunter OOSE, RDD, OMT, OBA, OODA, SOMA, MOSES und OPEN/OML.



Historie der objektorientierten Methoden und Notationen

Als Resultat dieser Bemühungen entstand die UML. Die Standardisierung, Pflege und Weiterentwicklung der Sprache wurde an die OMG übergeben, die die Sprache am 19. November 1997 als Standard akzeptierte.

Entstehungsgeschichte der Unified Modeling Language 2

Im August 1999 stieß die OMG die Entwicklung von UML2 an, indem sie einen entsprechenden Request for Information (RFI) publizierte.

Ein Jahr später, im September 2000, bat die OMG ihre Mitglieder und weitere interessierte Kreise um Vorschläge für UML2. Gemäß der neu für UML2 definierten Architektur hat die OMG drei Requests For Proposals (RFP) publiziert, einen UML 2.0 Infrastructure RFP [3], einen UML 2.0 Superstructure RFP [4] und einen UML 2.0 OCL RFP [5]. Wer Vorschläge einreichen wollte, hatte dazu ungefähr ein weiteres Jahr Zeit.

In einer ersten Runde haben verschiedene Gruppen und Einzelpersonen Entwürfe eingereicht. Mitte 2002 lagen von diesen Konsortien mehrmals überarbeitete und konsolidierte Antworten auf einzelne Request for proposals vor. Erst im Oktober 2002 konnten dann beim Treffen der zuständigen Arbeitsgruppe in Helsinki alle Vorschläge für die *UML 2.0 Infrastructure* und die *UML 2.0 Superstructure* präsentiert werden. Einen Termin für eine weitere überarbeitete Version der Vorschläge legte die Arbeitsgruppe auf Anfang Januar 2003 fest. Die Hauptschwierigkeit bestand nun darin, die unterschiedlichen Entwürfe zu *einer* Spezifikation zu verschmelzen. Einerseits wurde Kritik laut, dass sich die unterschiedlichen Philosophien in den eingereichten Vorschlägen nur schwerlich würden bereinigen lassen, andererseits reichte im Januar 2003 ein neues Konsortium unter dem Namen 4M einen Vorschlag (UML4MDA) ein, der die Differenzen zwischen den bisherigen Spezifikationen zu überbrücken versuchte.

Im März 2003 empfahl die zuständige Arbeitsgruppe die Vorschläge des Konsortiums U2 für die *UML 2.0 Infrastructure* und für die *UML 2.0 OCL* zur Freigabe, im Mai dann auch die *UML 2.0 Superstructure* des gleichen Konsortiums, so dass ab Juni 2003 drei *Finalization Task Forces* der OMG die Arbeit aufnehmen konnten, um die Teilspezifikationen abzuschließen. Die Task Forces konnten ihre Arbeit jedoch nicht wie geplant bis zum April 2004 abschließen und gründeten deshalb gleich anschließend eine zweite *Finalization Task Force*, die die verbleibenden Probleme bis zum September 2004 lösen sollte.

Im September 2004 konnten schließlich alle *Finalization Task Forces* ihre Arbeit beenden. Für die *UML 2.0 OCL* [6] und die *UML 2.0 Infrastructure* [7] lagen damit endgültig abgenommene Dokumente (*Final Adopted Specification*) vor. Nur bei der *UML 2.0 Superstructure* schien sich dieser letzte Schritt noch etwas zu verzögern: im März 2005 bot der OMG-Webauftritt weiterhin nur ein temporäres Dokument mit der informellen Bezeichnung *UML 2.0 Superstructure FTF convenience document* zum Herunterladen an.

Am 21. Oktober 2008 wurde die Beta 1 von UML Version 2.2 durch die OMG veröffentlicht, die wiederum im Februar 2009 in der finalen Version vorlag.[8] Neu hinzugekommen ist in der Version 2.2 das Profildiagramm, um eigendefinierte Stereotypen-Sammlungen strukturieren zu können.

Im Mai 2010 wurde UML 2.3 veröffentlicht. Diese Version hat vor allem Bugfixes am Metamodell und Schärfungen der Semantik von Modellelementen im Spezifikationsdokument der UML enthalten.[9]

Strukturierung

Der Umfang von UML ist während der Entwicklung von UML 1.0 bis zu UML2 laufend gewachsen. Sowohl für die Entwicklung von UML2 als auch für die Vermittlung, die Anwendung und nicht zuletzt für die Lesbarkeit der UML2-Spezifikation ist eine Strukturierung sehr wichtig. Was in den ersten Versionen von UML in einem Dokument spezifiziert werden konnte, musste deshalb für UML2 in Teilspezifikationen aufgeteilt werden. In den folgenden Abschnitten wird der Aufbau von UML2 beschrieben.

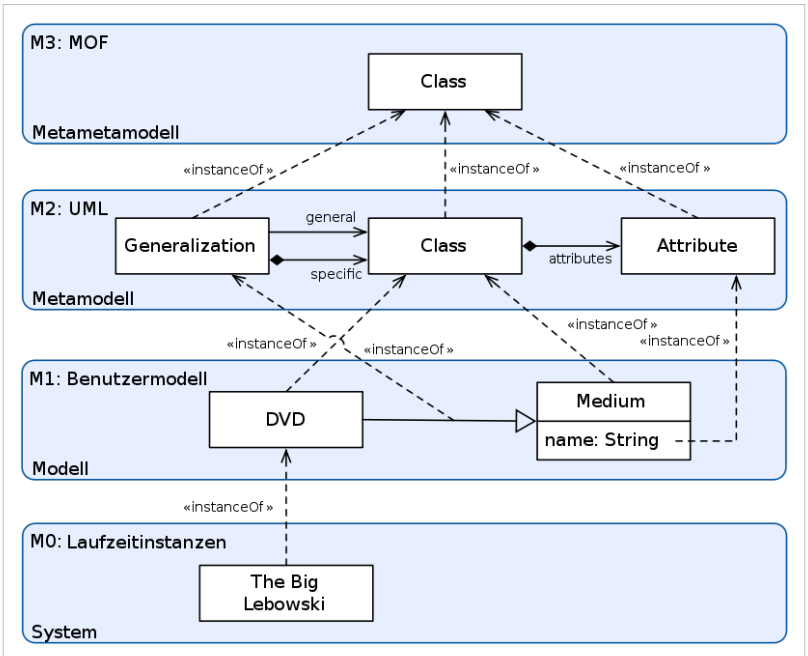
Teilspezifikationen

UML2 ist in drei Teilspezifikationen aufgeteilt. Die *UML 2.0 Infrastructure Specification* legt das Fundament für UML2, indem sie die am häufigsten verwendeten Elemente von UML2 und die Modellelemente beschreibt, die die restlichen Modellelemente spezialisieren. In diesem Dokument werden Konzepte wie die Klasse, die Assoziation oder die Multiplizität eines Attributs spezifiziert. Die *UML 2.0 Superstructure Specification* baut auf dem Fundament der UML 2.0 Infrastructure Specification auf und definiert die Modellelemente von UML2, die sich für bestimmte Einsatzzwecke eignen. Typische Konzepte, die in diesem Dokument spezifiziert werden, sind der Anwendungsfall, die Aktivität oder der Zustandsautomat. Schließlich spezifiziert das Dokument mit dem Titel *UML 2.0 Object Constraint Language* die Object Constraint Language 2.0 (OCL2).

Ein weiterer, vierter Teil beschäftigt sich nicht mit dem semantischen Modell von UML, sondern spezifiziert das Layout der Diagramme. Dieses Dokument trägt den Titel *UML 2.0 Diagram Interchange* und ist eine Neuerung in UML 2.0: UML 1.x kannte kein standardisiertes Format, mit dem das Layout von Diagrammen zwischen unterschiedlichen Werkzeugen ausgetauscht werden konnte. Die semantischen Informationen in einem UML-Modell konnte ein Werkzeug auch bisher an ein anderes Werkzeug übergeben, das Aussehen der Diagramme, das heißt die Positionen und Größe einzelner Diagrammelemente, ging dabei aber verloren. Diagram Interchange (DI) soll dieses Manko beseitigen.

Metamodellierung

Ähnlich wie sich natürliche Sprachen in Lexika oder Grammatiken selbst beschreiben, wurde auch UML als ein Sprachwerkzeug konzipiert, das sich mit einigen Sprachbestandteilen selbst erklärt.



Hierarchie der Metamodellierung

Mit der Meta Object Facility (MOF) werden Modellelemente von UML2 spezifiziert und dadurch z. B. mit dem Format Meta Interchange XMI austauschbar. UML ist dazu in vier Schichten *M0* bis *M3* gegliedert. Die bereits erwähnte MOF (*M3*) stellt eine der vier Schichten dar. Es ist die Metasprache der Metasprachen von UML2 und beinhaltet deren grundlegende Elemente. Die Metasprache von UML2 (*M2*) spezifiziert deren grundlegende Elemente mit ihren Eigenschaften. Die in der Praxis verwendete UML2 befindet sich auf der Ebene *M1*. Damit werden die Objekte der *M0*-Schicht dargestellt. Dies sind die konkreten Laufzeitinstanzen des Systems.

Wie in UML2.0, war auch UML1.x auf der dritten von vier Metamodellierungsebenen eingeordnet. Zu UML 1.x besteht jedoch ein wesentlicher Unterschied: Die auf diesen vier Ebenen verwendeten Modellierungssprachen, besonders die Sprachen auf den Ebenen M2 und M3, teilen sich die gemeinsame Spracheinheit der *Infrastrukturbibliothek* (*InfrastructureLibrary*). Sie wird in der *UML 2.0 Infrastructure* definiert und bildet einen Kern der grundlegenden Modellierungselemente, der sowohl in der *UML 2.0 Infrastructure* als auch in der *UML 2.0 Superstructure* und in der *MOF 2.0* eingesetzt wird.

Spracheinheiten

Die *UML 2.0 Superstructure* ist auf einer ersten Ebene modular in Spracheinheiten (engl. *language units*) aufgebaut. Eine Spracheinheit umfasst eine Menge von eng zusammenhängenden Modellierungselementen, mit denen ein Benutzer einen ausgewählten Aspekt eines Systems mit einem bestimmten Formalismus modellieren kann. Die Spracheinheit *Aktivitäten* (engl. *Activities*) umfasst zum Beispiel Elemente für die Modellierung eines Systemverhaltens, das sich am besten mit dem Formalismus von Daten- und Kontrollflüssen darstellen lässt.

Einteilung der Spracheinheiten in Schichten

Auf einer dritten Stufe sind die meisten Spracheinheiten in mehrere Schichten (engl. *Compliance Level*) gegliedert. Die unterste Schicht umfasst jeweils die einfachsten und am häufigsten verwendeten Modellierungselemente, während höhere Schichten zunehmend komplexere Modellierungselemente einführen. Die Spracheinheit Aktivitäten umfasst beispielsweise *FundamentalActivities* als unterste Schicht und darauf aufbauend die Schicht *BasicActivities*. *FundamentalActivities* definiert zunächst nur, dass Aktivitäten strukturell aus hierarchisch geschachtelten Gruppen von Aktionen bestehen. *BasicActivities* erweitert dieses Gerüst um Kanten und weitere Hilfsknoten zu einem Graphen, den man in UML2 dann visuell als Aktivitätsdiagramm darstellt.

Spracheinheiten

Aktionen

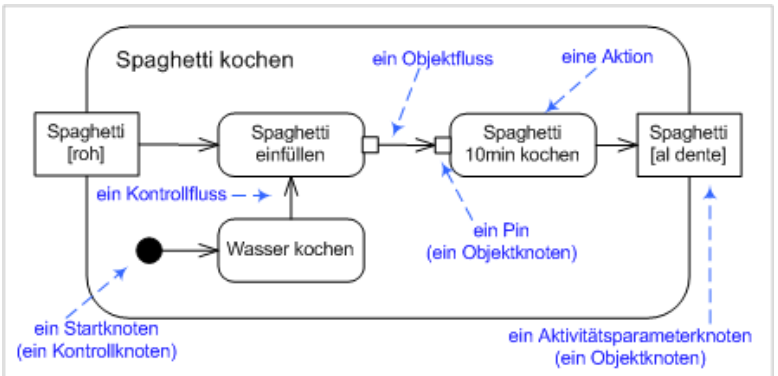
Die Spracheinheit *Aktionen* (engl. *actions*) umfasst die Definition der Aktionen in UML2. Aktionen sind die elementaren Bausteine für die Modellierung eines Verhaltens. Sie können Eingabewerte über sogenannte Eingabepins entgegennehmen bzw. Ausgabewerte an sogenannten Ausgabepins produzieren.

UML2 definiert in dieser Spracheinheit mehrere Gruppen von grundlegenden Aktionen, siehe Aktion.

Beispiel der graphischen Notation für eine Aktion mit zwei Eingabe- und einem Ausgabepin

Aktivitäten

Die Aktivität ist das zentrale Element, das in der Spracheinheit *Aktivitäten* definiert ist. Sie ist gleichzeitig eine der wesentlichen Neuerungen von UML2 gegenüber UML 1.4. Eine Aktivität ist ein Modell für ein Verhalten, das als Menge von elementaren Aktionen beschrieben ist, zwischen denen Kontroll- und Datenflüsse existieren. Aus der Sicht eines Softwaresystems würde ein Aktivitätsdiagramm das dynamische Verhalten des entsprechenden Systems darstellen.



Spaghetti-Kochen modelliert als Aktivität

Aktivitäten haben die Struktur eines Graphen. Knoten stellen Aktionen dar sowie Punkte, an denen die Flüsse zwischen den Aktionen kontrolliert werden; Kanten stehen für Objekt- und Kontrollflüsse. Die Aufgabe des Sprachpakets Aktivitäten ist es, alle Typen von Knoten und Kanten zu definieren, die für die Modellierung von

Aktivitäten benötigt werden.

Knoten werden in Objekt- und Kontrollknoten unterschieden, Kanten analog dazu in Objekt- und Kontrollflüsse.

Komplexere Aktivitäten können verschachtelt und mit Kontrollstrukturen modularisiert werden.

Graphisch werden Aktivitäten in Aktivitätsdiagrammen modelliert.

Allgemeines Verhalten

Die Spracheinheit *Allgemeines Verhalten* umfasst die allgemeinen Modellelemente für die Spezifikation des Verhaltens eines mit UML2 modellierten Systems. Hier sind die Modellelemente zusammengefasst, die für die Spezifikation von Aktivitäten, Interaktionen oder Zustandsautomaten benötigt werden.

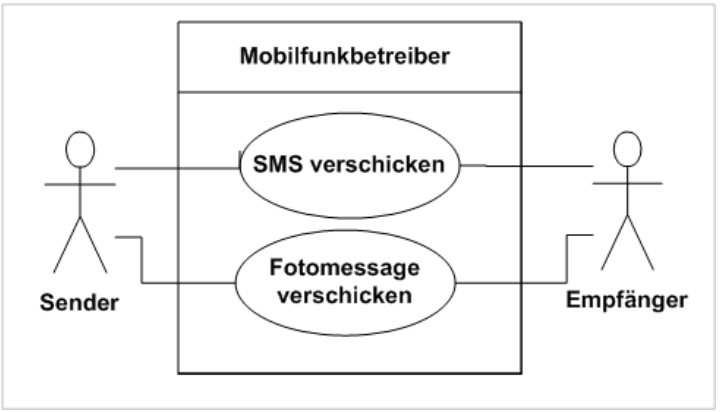
Die Spracheinheit definiert die Gemeinsamkeiten jeder Verhaltensbeschreibung und dass eine aktive Klasse ein eigenes Verhalten haben kann. Verhalten in einem System, das mit UML2 modelliert ist, startet immer aufgrund von diskreten Ereignissen. Dieses Sprachpaket definiert, welche Arten von Ereignissen UML2 unterstützt.

Die Behandlung der Zeit wird ebenfalls weitgehend in diesem Sprachpaket geregelt. Es definiert Metaklassen für die Beobachtung der Zeit (*TimeObservationAction*), für die Notation von Zeitausdrücken (*TimeExpression*), für die Definition von Zeitintervallen (*TimeInterval*) sowie für das Konzept einer zeitlichen Dauer (*Duration*).

Anwendungsfälle

Die Spracheinheit *Anwendungsfälle* (engl. *use case*) stellt Elemente für die Modellierung von Anforderungen an ein System zur Verfügung. Das wichtigste Element ist der Anwendungsfall. Anwendungsfälle halten fest, was ein System tun soll. Das zweite wichtige Element ist der Akteur. Akteure spezifizieren, *wer* (im Sinne einer Person) oder *was* (im Sinne eines anderen Systems) etwas mit dem System tun soll.

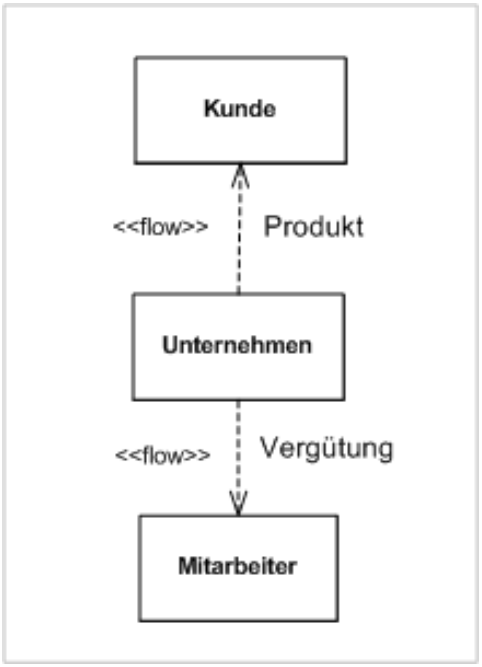
Graphisch werden Anwendungsfälle in Anwendungsfalldiagrammen dargestellt.



Beispiel für die graphische Darstellung zweier Anwendungsfälle und zweier Akteure

Informationsflüsse

Den Techniken, die UML2 für die Spezifikation des Verhaltens eines Systems anbietet, liegen präzise semantische Modelle zugrunde. Das gilt insbesondere für Verhaltensbeschreibungen mit Hilfe von Interaktionen oder Aktivitäten, die zudem darauf ausgerichtet sind, das Verhalten eines Systems sehr feingranular zu spezifizieren. Soll das Modell eines Systems nur einige grundlegende Informationsflüsse im System aufzeigen, eignen sich diese Techniken deshalb nur bedingt.



Beispiel eines Strukturdiagramms mit zwei Informationsflüssen

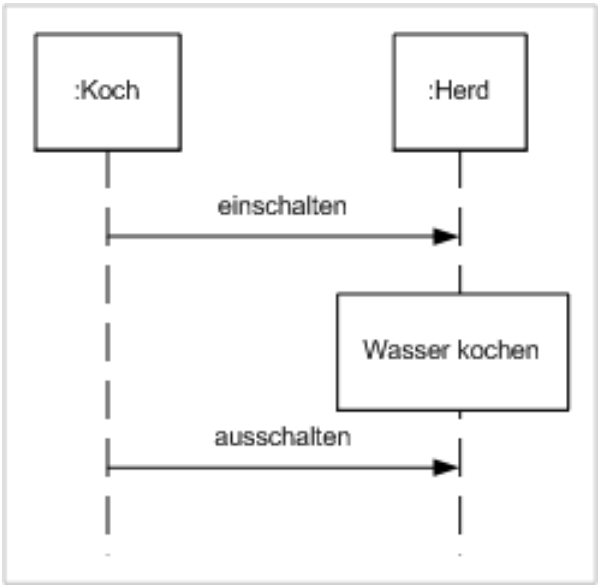
Die Spracheinheit *Informationsflüsse*, die in UML2 neu eingeführt wurde, stellt Modellelemente zur Verfügung, um diese Situation zu verbessern. Sie bietet die Modellelemente Informationseinheit und Informationsfluss an, mit denen ein Modellierer Informationsflüsse in einem System auf hoher Abstraktionsstufe festhalten kann.

Informationsflüsse können dabei eine Vielzahl von anderen Modellelementen von UML2 verbinden, insbesondere Klassen, Anwendungsfälle, Ausprägungsspezifikationen, Akteure, Schnittstellen, Ports und noch einige mehr.

UML2 gibt keinen Diagrammtyp für Informationsflüsse vor. Die graphische Notation für Informationsflüsse und Informationseinheiten kann in allen Strukturdiagrammen vorkommen.

Interaktionen

Das Verhalten eines modellierten Systems kann in UML2 auf unterschiedliche Art und Weise spezifiziert werden. Eine davon ist die Modellierung von Interaktionen. Eine Interaktion ist die Spezifikation eines Verhaltens, das am Besten über den Austausch von Meldungen zwischen eigenständigen Objekten beschrieben wird. Die Spracheinheit stellt dafür die geeigneten Modellelemente zur Verfügung.



Beispiel für die Spezifikation einer Interaktion mit Hilfe eines Sequenzdiagramms

Wer Interaktionen modelliert, geht davon aus, dass das modellierte System aus einem Netzwerk von Objekten besteht, die untereinander Meldungen austauschen. Schickt ein Objekt einem anderen Objekt eine Meldung, kann man zwei Ereignisauftritte identifizieren: erstens das Auftreten eines Meldungsereignisses, wenn die Meldung vom ersten Objekt abgeschickt wird sowie zweitens eines Meldungsereignisses, wenn die Meldung beim zweiten Objekt ankommt. Weitere Ereignisse treten auf, wenn eine Aktion oder ein anderes Verhalten im Kontext eines Objekts beginnt oder endet. Eine *Spur* (*trace*) bezeichnet

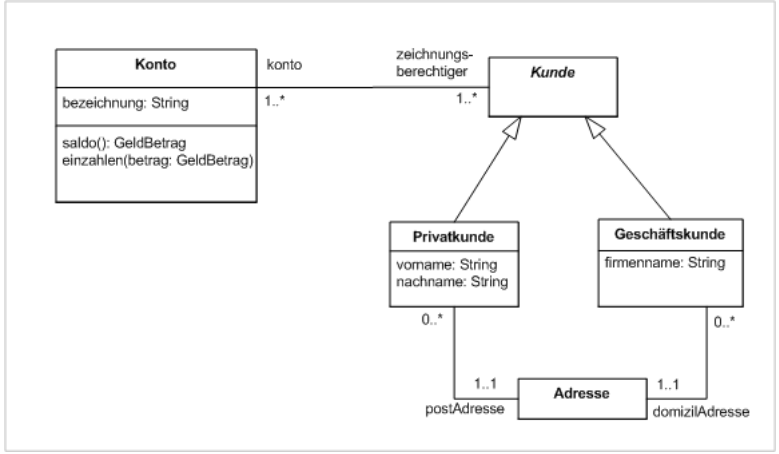
eine Folge von solchen Ereignissen. Interaktionen spezifizieren nun ein Verhalten als zwei Mengen von Spuren, einer Menge *gültiger* und einer Menge *ungültiger* Spuren.

Damit ist präziser gesagt, was wir meinen, wenn wir von Interaktionen sprechen: die *Bedeutung* (Semantik) einer Interaktion ist durch Mengen von Spuren gegeben. *Modelliert* werden Interaktionen jedoch als Mengen von Lebenslinien, auf denen Aktionen und andere Verhaltensweisen ablaufen und zwischen denen Nachrichten ausgetauscht werden.

Interaktionen modelliert man graphisch in Kommunikationsdiagrammen, in Sequenzdiagrammen oder in Zeitverlaufsdiagrammen.

Klassen

Die Spracheinheit *Klassen* umfasst den eigentlichen Kern der Modellierungssprache. Sie definiert insbesondere, was man in UML2 unter einer Klasse versteht und welche Beziehungen zwischen Klassen möglich sind. In dieser Spracheinheit sind grundlegende Prinzipien von UML2 definiert. Die Metaklasse *Element* ist das Wurzelelement für alle anderen Modellelemente. Jedes Element kann andere Elemente besitzen, auch beliebig viele Kommentare, die wiederum andere Elemente kommentieren können. Zwischen Elementen können Beziehungen definiert werden. Elemente können benannt sein und gehören in diesem Fall zu einem Namensraum. Weiter können gewisse Elemente einen Typ haben. Sie werden dann als *typisierte Elemente* bezeichnet. Einem Element kann eine Multiplizität mit einer unteren und einer oberen Schranke zugeordnet sein.



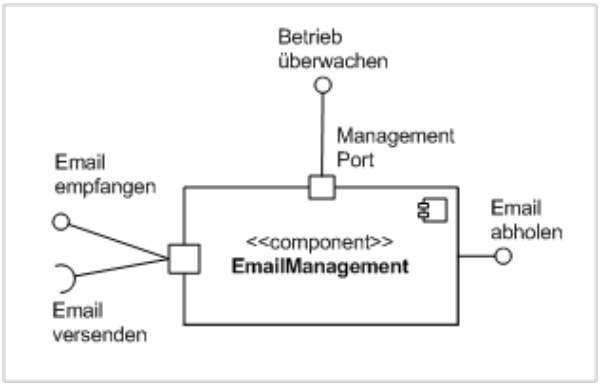
Beispiel für ein Klassendiagramm, das Elemente aus der Spracheinheit *Klassen* verwendet

Diese Spracheinheit enthält vier Unterpakete. Das Unterpaket *Kernel* umfasst zentrale Modellierungselemente, die aus der *UML 2.0 Infrastructure* wiederverwendet werden. Dazu gehören die Klasse, die Ausprägungsspezifikation, der Namensraum, das Paket, das Attribut, die Assoziation, die Abhängigkeitsbeziehung, der Paketimport, die Paketverschmelzung und die Generalisierung. Das zweite Unterpaket, *AssociationClasses*, umfasst die Definition von Assoziationsklassen. *Interfaces*, das dritte Unterpaket, stellt die Definition von Schnittstellen bereit. Schließlich deklariert das Unterpaket *PowerTypes* Modellelemente für die sogenannten PowerTypes.

Elemente aus dieser Spracheinheit werden meistens in Klassendiagrammen, Objektdiagrammen und Paketdiagrammen dargestellt.

Komponenten

Komponenten sind modulare Teile eines Systems, die so strukturiert sind, dass sie in ihrer Umgebung durch eine andere, äquivalente Komponente ersetzt werden könnten. In der Softwareentwicklung verwendet man insbesondere das Konzept der Softwarekomponente, um ein Softwaresystem in modulare Teile zu gliedern. Die Spracheinheit *Komponenten* von UML2 stellt Konstrukte zur Verfügung, um Systeme, die aus Komponenten aufgebaut sind, zu modellieren.



Beispiel einer Komponente mit drei angebotenen und einer benötigten Schnittstelle, sowie zwei Ports

Das wichtigste Element ist die Komponente, die eine innere Struktur gegen außen abgrenzt. Die Spezifikation einer Komponente deklariert vor allem den von außen sichtbaren Rand und definiert damit eine Black-Box-Sicht auf die Komponente. Sichtbar sind eine Menge von angebotenen und erforderlichen Schnittstellen sowie allenfalls eine Menge von Ports.

Die Spracheinheit umfasst ferner den Delegations- und den Kompositionskonnektor. Der Delegationskonnektor verbindet Ports auf der Hülle einer Komponente mit Elementen im Innern der Komponente. Der Kompositionskonnektor verbindet angebotene Schnittstellen einer Komponente mit benötigten Schnittstellen einer anderen Komponente.

Die Elemente dieser Spracheinheit werden meistens in Komponentendiagrammen, zum Teil aber auch in Klassendiagrammen oder Verteilungsdiagrammen dargestellt.

Kompositionsstrukturen

Die Spracheinheit *Kompositionsstrukturen* bereichert UML2 um einen neuen Ansatz für die Modellierung der inneren Struktur eines zusammengesetzten Ganzen. Das „Ganze“ wird dabei als gekapselter Classifier modelliert, für die „Teile“ stellt diese Spracheinheit die Parts zur Verfügung. Untereinander können Parts durch Konnektoren verbunden sein. Der gekapselte Classifier steht also für ein System mit klarer Abgrenzung von Innen und Außen, dessen innere Struktur mit Hilfe von Parts und Konnektoren spezifiziert ist. Damit die Grenze zwischen Innen und Außen zumindest teilweise durchlässig ist, kann der gekapselte Classifier auf der Hülle über eine Menge von Ein- und Ausgangspunkten, so genannten Ports, verfügen.

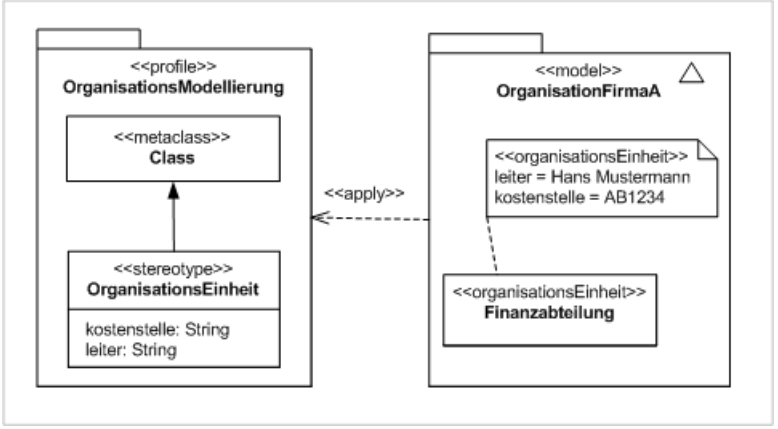
Elemente aus dieser Spracheinheit werden meistens in Kompositionsstrukturdiagrammen dargestellt.

Modelle

Die Spracheinheit *Modelle* umfasst nur ein Modellelement: das Modell.

Profile

UML2 stellt mit der Spracheinheit *Profile* einen leichtgewichtigen Erweiterungsmechanismus zur Verfügung, mit dem sie spezifischen Einsatzgebieten angepasst werden kann. Der Mechanismus wird als leichtgewichtig bezeichnet, weil er das Metamodell von UML2 unverändert lässt, oft ein entscheidender Vorteil, denn auf dem Markt erhältliche Werkzeuge für die Erstellung und Pflege von UML2-Modellen können oft nur mit Modellen basierend auf dem standardisierten UML2-Metamodell umgehen.



Beispiel für die Definition und die Anwendung eines vereinfachten Profils für die Organisationsmodellierung

UML2 umfasst in ihren Spracheinheiten verschiedene Möglichkeiten für die Modellierung der Struktur und des Verhaltens eines Systems, muss dabei aber auf einer generischen Ebene bleiben. Sie verwendet zum Beispiel die generischen Begriffe Aktivität oder Artefakt und kennt den spezifischen Begriff Geschäftsprozess aus der Geschäftsmodellierung oder Enterprise Java Beans der Java-Plattform nicht. Falls diese Begriffe in der Modellierung benötigt werden, müssen sie über den Erweiterungsmechanismus der Profile zu UML2 hinzugefügt werden. Auch spezielle Notationen, zum Beispiel eine realistischere Zeichnung anstelle des Strichmännchens, das einen Akteur darstellt, können UML2 mit Hilfe der Profile hinzugefügt werden. Weiter können Profile Lücken in der Definition der Semantik von UML2 schließen, die dort absichtlich für spezifische Einsatzgebiete offen gelassen wurden (engl. *semantic variation points*). Schließlich können Profile Einschränkungen definieren, um die Art und Weise zu beschränken, wie ein Element aus UML2 verwendet wird.

Mit Hilfe des Erweiterungsmechanismus der Profile kann das Metamodell von UML2 jedoch nicht beliebig angepasst werden. So können zum Beispiel keine Elemente aus dem Metamodell von UML2 entfernt, keine Einschränkungen aufgehoben und keine echten neuen Metaklassen, sondern nur Erweiterungen (Stereotypen) von bestehenden Metaklassen, deklariert werden.

Die Spracheinheit definiert zunächst das Konzept eines Profils. Ein Profil ist eine Sammlung von Stereotypen, und definiert die eigentliche Erweiterung. Ein Profil *ist* ein Paket und wird auf andere Pakete *angewendet*, womit die Erweiterung, die das Profil definiert, für das entsprechende Paket gilt.

UML2 kennt seit der UML 2.2 einen speziellen Diagrammtyp für Profile. Die graphischen Notationen für Elemente aus dieser Spracheinheit kommen sowohl in Paketdiagrammen als auch in Klassendiagrammen vor.

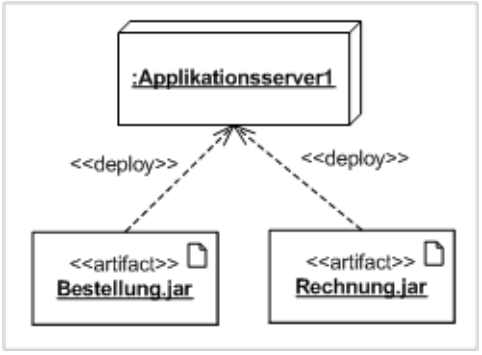
Schablonen

Die Spracheinheit Schablonen (engl. *Templates*) umfasst Modellelemente für die Parametrisierung von Klassifizierern, Klassen und Paketen.

Verteilungen

Die Spracheinheit *Verteilungen* (engl. *Deployments*) ist auf ein sehr spezifisches Einsatzgebiet ausgerichtet, nämlich auf die Verteilung von lauffähiger Software in einem Netzwerk. UML2 bezeichnet eine so installierbare Einheit als Artefakt und geht davon aus, dass diese auf Knoten installiert werden. Knoten können entweder Geräte oder Ausführungsumgebungen sein. Eine Verteilungsbeziehung, das heißt eine spezielle Abhängigkeitsbeziehung, modelliert, dass ein Artefakt auf einem Knoten installiert wird.

Elemente aus dieser Spracheinheit werden normalerweise in Verteilungsdiagrammen dargestellt.

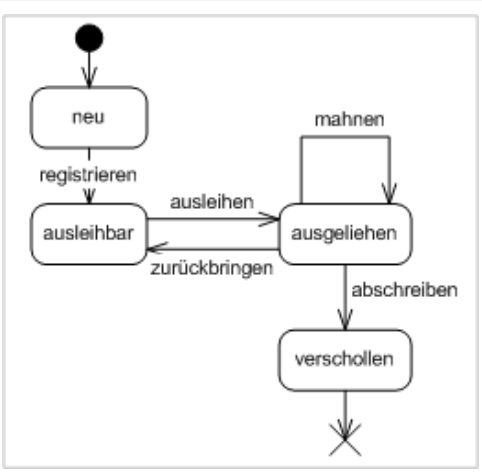


Beispiel eines Verteilungsdiagramms mit einem Knoten und zwei Artefakten

Zustandsautomaten

Die Spracheinheit *Zustandsautomaten* (engl. *state machines*) umfasst Modellelemente, die für die Modellierung von Zustandsautomaten eingesetzt werden.

UML2 setzt Zustandsautomaten in erster Linie für die Spezifikation des Verhaltens eines Systems ein. So kann ein Zustandsautomat zum Beispiel das Verhalten der Instanzen einer aktiven Klasse modellieren. Zusätzlich können Zustandsautomaten aber auch eingesetzt werden, um eine zulässige Nutzung einer Schnittstelle oder eines Ports zu spezifizieren. Der Zustandsautomat modelliert dabei zum Beispiel, in welcher Reihenfolge Operationen einer Schnittstelle aufgerufen werden dürfen. Im ersten Fall spricht man von einem Verhaltenszustandsautomaten, im zweiten von einem Protokollzustandsautomaten.



Beispiel eines Zustandsautomaten für die Zustände eines Buchs in einer öffentlichen Bibliothek

Zustandsautomaten werden graphisch in Zustandsdiagrammen dargestellt. Diese sind in UML eine Variante der klassischen Statecharts.

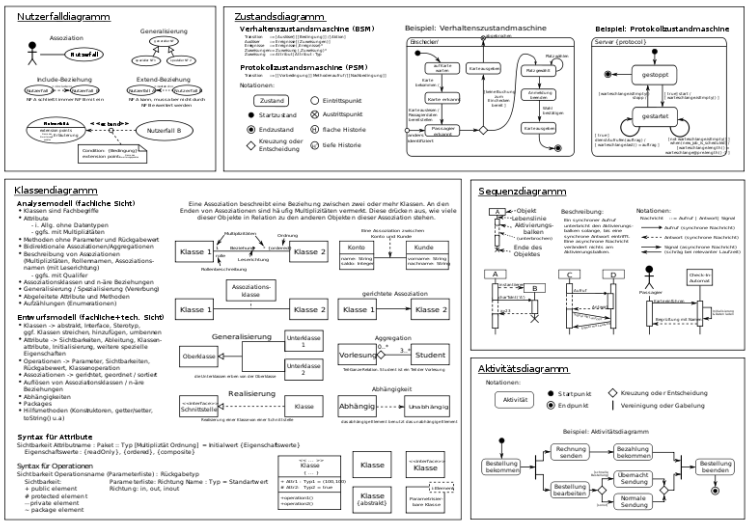
Darstellung in Diagrammen

UML2 kennt sieben Strukturdiagramme:

- das Klassendiagramm,
- das Kompositionsstrukturdiagramm (auch: Montagediagramm),
- das Komponentendiagramm,
- das Verteilungsdiagramm,
- das Objektdiagramm,
- das Paketdiagramm und
- das Profildiagramm.

Dazu kommen sieben Verhaltensdiagramme:

- das Aktivitätsdiagramm,
- das Anwendungsfalldiagramm (auch: Use-Case o. Nutzfalldiagramm genannt),
- das Interaktionsübersichtsdiagramm,
- das Kommunikationsdiagramm,
- das Sequenzdiagramm,
- das Zeitverlaufsdiagramm und
- das Zustandsdiagramm.



UML Diagrammübersicht

Die Grenzen zwischen den vierzehn Diagrammtypen verlaufen weniger scharf, als diese Klassifizierung vermuten lässt. UML2 verbietet nicht, dass ein Diagramm graphische Elemente enthält, die eigentlich zu unterschiedlichen Diagrammtypen gehören. Es ist sogar denkbar, dass Elemente aus einem Strukturdiagramm und aus einem Verhaltensdiagramm auf dem gleichen Diagramm dargestellt werden, wenn damit eine besonders treffende Aussage zu einem Modell erreicht wird.

UML2 geht aber in anderer Hinsicht weit formaler mit Diagrammen um als UML 1.4. Neu definiert UML2 unter dem Namen *UML 2.0 Diagram Interchange* ein Austauschformat für Diagramme, so dass unterschiedliche Werkzeuge, mit denen Modelle basierend auf UML2 erstellt werden, die Diagramme austauschen und wiederverwenden können. In UML 1.x war das nur für die Repository-Modelle *hinter* den Diagrammen möglich, aber nicht für die eigentlichen Diagramme.

Erstellen von Diagrammen

Siehe dazu auch UML-Werkzeug

Diagramme von UML2 können auf verschiedene Arten erstellt werden. Wenn die Notation von UML2 als gemeinsame Sprache eingesetzt wird, um in einem Analyseteam Entwürfe von Analysemodellen an der Weißwandtafel (*Whiteboard*) festzuhalten, reichen Stifte und Papier als Werkzeug. Häufig werden Diagramme von UML2 jedoch mit Hilfe von speziellen Programmen (UML-Werkzeugen) erstellt, die man in zwei Klassen einteilen kann.

Programme in der ersten Gruppe helfen beim Zeichnen von Diagrammen der UML2, ohne dass sie die Modellelemente, welche den graphischen Elementen auf den Diagrammen entsprechen, in einem Repository ablegen. Zu dieser Gruppe gehören alle Programme zum Erstellen von Zeichnungen.

Die zweite Gruppe besteht aus Programmen, die die Erstellung von Modellen und das Zeichnen von Diagrammen von UML2 unterstützen.

Austausch von Modellen und Diagrammen

Damit Modelle von einem Werkzeug an andere übergeben werden können, definiert die Object Management Group ein standardisiertes Austauschformat, das auch für UML-Modelle eingesetzt wird. Das Format basiert auf der Auszeichnungssprache XML und heißt XML Metadata Interchange (XMI). Die Grundlage für die Austauschbarkeit ist das MOF, auf dessen Konzept beide Sprachen, XMI und UML, beruhen.

Für die UML-Versionen 1.x sah das Format keine Möglichkeit vor, Diagramme in einem standardisierten Format auszutauschen, was von vielen Anwendern als wesentliche Lücke wahrgenommen wurde. Parallel zur Entwicklung von UML2 hat die OMG deshalb auch das standardisierte Austauschformat XMI überarbeitet. Unter anderem wurde die beschriebene Lücke geschlossen, indem die Spezifikation unter dem Namen *UML 2.0 Diagram Interchange* um ein Format für den Austausch von Diagrammen erweitert wurde.

Ein Austausch mit anderen Modellierungssprachen ist auch mittels Modell-zu-Modell-Transformation möglich. Dazu hat die OMG den Standard MOF QVT definiert. Im Gegensatz zu einem reinen Austauschformat kann eine Transformation auch eine eigene Semantik enthalten. So kann zum Beispiel festgelegt werden, wie ein Klassenmodell auf ein ER-Modell abzubilden ist. Die damit einhergehende Flexibilität ist insbesondere auch beim Austausch zwischen Werkzeugen nützlich, da verschiedene Anbieter praktisch immer auch individuelle Varianten des UML-Metamodells haben.

Siehe auch

- Rational Unified Process
- Round Trip Engineering
- Verwandte OMG-Initiativen und Sprachempfehlungen:
  - Model Driven Architecture (MDA)
  - MOF 2 Query/Views/Transformations (QVT)
  - Systems Modeling Language (SysML)
  - Common Warehouse Metamodel (CWM)
  - Object Constraint Language (OCL)
  - Modeling and Analysis of Real-time and Embedded systems (MARTE)
- Weitere Modellierungsmethoden und Sprachfamilien:
  - IDEF
  - EXPRESS (siehe STEP)
  - Fundamental Modeling Concepts (FMC)

Einzelnachweise

[1] http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/Teil 1 der Spezifikation der Sprache (Infrastruktur)
[2] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML Webseite der OMG
[3] http://www.omg.org/docs/ad/00-09-01.pdf
[4] http://www.omg.org/docs/ad/00-09-02.pdf
[5] http://www.omg.org/docs/ad/00-09-03.pdf
[6] http://www.omg.org/cgi-bin/doc?ptc/2003-10-14
[7] http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf
[8] www.omg.org/spec/UML/2.2/ (http://www.omg.org/spec/UML/2.2/)
[9] "UML in Version 2.3 ist fertig" (http://www.heise.de/developer/UML-in-Version-2-3-ist-fertig--/blog/artikel/143568) Bernds Management-Welt auf heise Developer

Literatur

- Heide Balzert: *UML 2 in 5 Tagen*, W3L, 2005, ISBN 3-937137-61-0
- H. Baumann, P. Grässle, Ph. Baumann, *UML 2 projektorientiert*, Galileo Computing, 2007, ISBN 978-3-8362-1014-0
- Grady Booch, James Rumbaugh, Ivar Jacobson: *Das UML-Benutzerhandbuch.* Addison-Wesley, 1999, ISBN 3-8273-1486-0
- M. Born, E. Holz, O. Kath: *Softwareentwicklung mit UML 2*, Addison-Wesley, 2004, ISBN 3-8273-2086-0
- B. Brügge, A. H. Dutoit: *Objekt-orientierte Softwaretechnik mit UML, Entwurfsmustern und Java*, Pearson Studium, 2004, ISBN 3-8273-7082-5
- M. Fowler; Kendall, Scott: *UML konzentriert*, Addison-Wesley Verlag, 2000, ISBN 3-8273-1617-0
- M. Fowler: *UML Distilled*, 3. Auflage, Addison-Wesley, 2003, ISBN 0-321-19368-7
- M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger: *UML@Work*, Dpunkt Verlag, 2005, ISBN 3-89864-261-5
- B. Kahlbrandt: *Software Engineering mit der Unified Modeling Language*, Springer, 2001, ISBN 3-540-41600-5
- Christoph Kecher: *UML 2 – Das umfassende Handbuch*, Galileo Computing, 2009, ISBN 978-3-8362-1419-3
- Urs B. Meyer, Simone E. Creux, Andrea K. Weber Marin: *Grafische Methoden der Prozessanalyse*, Hanser Verlag, ISBN 3-446-40041-9
- B. Oestereich: *Analyse und Design mit UML 2.3: Objektorientierte Softwareentwicklung*, Oldenbourg Verlag, 2009, ISBN 978-3-486-58855-2
- Dan Pilone: *UML – kurz & gut*, O'Reilly, ISBN 3-89721-263-3
- B. Rumpe, *Modellierung mit UML*, Springer Verlag, 2004, ISBN 3-540-20904-2
- B. Rumpe, *Agile Modellierung mit UML*, Springer Verlag, 2004, ISBN 3-540-20905-0
- Chris Rupp, Stefan Queins, Barbara Zengler: *UML 2 glasklar. Praxiswissen für die UML-Modellierung*, Hanser Verlag, 2007, ISBN 978-3-446-41118-0
- Sinan Si Alhir: *Learning UML*, O'Reilly, ISBN 0-596-00344-7
- H. Störrle: *UML 2 für Studenten*, Pearson Studium Deutschland, 2005, ISBN 3-8273-7143-0
- H. Störrle: *UML 2 erfolgreich einsetzen*, Addison-Wesley, 2005, ISBN 3-8273-2268-5
- T. Weilkiens, B. Oestereich: *UML2-Zertifizierung*, Dpunkt Verlag, 2004, ISBN 3-89864-294-1
- T. Weilkiens, B. Oestereich: *UML2-Zertifizierung: Intermediate Stufe*, Dpunkt Verlag, 2005, ISBN 3-89864-312-3
- T. Weilkiens, B. Oestereich: *UML 2.0 Zertifizierungsvorbereitung. Fundamental, Intermediate und Advanced*, Dpunkt Verlag, 2006, ISBN 3-89864-424-3
- W. Zuser, T. Grechenig, M. Köhle: *Software Engineering mit UML und dem Unified Process*, Pearson Studium, 2004, ISBN 3-8273-7090-6

Weblinks

Spezifikationen zur UML2

- UML 2.3 Infrastructure Specification (http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/) (englisch) (PDF)
- UML 2.3 Superstructure Specification (http://www.omg.org/spec/UML/2.3/Superstructure/PDF/) (englisch) (PDF)
- UML Specification in allen Versionen (http://www.omg.org/spec/UML/)

Weitere

- *MOF 2.0 Core Specification* (http://www.omg.org/spec/MOF/2.0/PDF/) (englisch) (PDF)
- *MOF 1.4 Specification* (http://www.omg.org/spec/MOF/1.4/PDF/) (englisch) (PDF)
- Website der OMG zur UML (http://www.uml.org/) (englisch)
- Der moderne Softwareentwicklungprozess mit UML (http://www.highscore.de/uml/) Online-Buch zur UML (deutsch)
- UML Referenzkarte (http://www.oio.de/public/objektorientierung/uml-referenz-2-1/index.htm)
- Notationsübersicht (http://www.oose.de/downloads/uml-2-Notationsuebersicht-oose.de.pdf) (PDF, 304 kB).
- Death by UML Fever (http://queue.acm.org/detail.cfm?id=984495) – Ein kritischer Artikel über UML, erschienen in den Communications of the ACM (http://de.wikipedia.org/wiki/Communications_of_the_ACM)
- Der UML Tool Guide (http://www.entracons.de/uml-tool-guide.html) – Der UML Tool Guide enthält UML-Werkzeuge nach Zielgruppen sortiert. Viele Begriffe werden im Glossar erklärt. (deutsch)

Pflichtenheft

Das **Pflichtenheft** beschreibt in konkreterer Form, wie der Auftragnehmer die Anforderungen im Lastenheft zu lösen gedenkt – das sogenannte *wie und womit*.

Unterscheidung zum Lastenheft

Ein wesentlicher Unterschied zu einem Lastenheft besteht darin, dass das Pflichtenheft dem Auftragnehmer gehört. Ein Lastenheft beschreibt die Gesamtheit der Forderungen des Auftraggebers, im Pflichtenheft ist in konkreterer Form beschrieben, wie der Auftragnehmer die Anforderungen im Lastenheft zu lösen gedenkt.

Andere Begriffe

Neben dem Begriff *Pflichtenheft* findet man in der Praxis auch unscharfe Bezeichnungen wie *Fachspezifikation*, *fachliche Spezifikation*, *Fachfeinkonzept*, *Sollkonzept*, *Funktionelle Spezifikation*, *Gesamtsystemspezifikation*, *Implementierungsspezifikation* oder *Feature Specification*. Da diese Bezeichnungen in der Regel nicht standardisiert sind, können damit durchaus Dokumente im Sinne des Pflichtenhefts gemeint sein, aber auch Fachkonzept, Lastenheft oder etwas anderes.

Definition

Laut DIN 69901-5 umfasst das Pflichtenheft die „vom Auftragnehmer erarbeiteten Realisierungsvorgaben aufgrund der Umsetzung des vom Auftraggeber vorgegebenen Lastenhefts“. Die Anforderungen des zuvor ausgearbeiteten Lastenhefts sind nun mit technischen Festlegungen der Betriebs- und Wartungsumgebung verknüpft.

Nach VDI-Richtlinie 2519 Blatt 1 ist das Pflichtenheft die Beschreibung der Realisierung aller Kundenanforderungen, die im Lastenheft gefordert werden.

Das Pflichtenheft wird vom Auftragnehmer formuliert und auf dessen Wunsch vom Auftraggeber bestätigt. Idealerweise sollten erst nach dieser Bestätigung die eigentlichen Entwicklungs-/Implementierungsarbeiten beginnen. Der Auftragnehmer hat einen durch den Vertrag bestimmten Anspruch auf solche Bestätigung (Mitwirkungspflicht nach §643 BGB).

Praxis

Es ist bewährte Praxis, bei der Erstellung eines Pflichtenheftes das Ein- und Ausschlussprinzip zu verwenden, das heißt, konkrete Fälle explizit ein- oder auszuschließen.

Bei Lieferung wird formell eine Abnahme vollzogen, die die Ausführung des Werkvertrages oder auch des Kaufvertrages beschließt. Diese Abnahme wird häufig über einen Akzeptanztest ausgeführt, der feststellt, ob die Forderungen des Lastenheftes in dem Verständnis des Bestellers erfüllt wurden.

In der Softwareentwicklung wird das Pflichtenheft unter anderem im V-Modell 97 definiert. Im aktuellen V-Modell XT wurde die Bezeichnung in *Gesamtsystemspezifikation (Pflichtenheft)* geändert. Für internationale Projekte wird heutzutage stattdessen meistens eine Software Requirements Specification, welche die Inhalte des Lasten- und Pflichtenhefts enthält, erstellt.

Aufbau

Pflichtenhefte sind für Projekte, Geräte oder Aggregate, aber auch für ganze Anlagen üblich. Im Folgenden eine beispielhafte Gliederung für ein Projekt aus der Informationstechnik.

Ein Pflichtenheft sollte wie folgt gegliedert sein[1] :

1. Zielbestimmung
    1. Musskriterien: für das Produkt unabdingbare Leistungen, die in jedem Fall erfüllt werden müssen
    2. Sollkriterien: die Erfüllung dieser Kriterien wird angestrebt
    3. Kannkriterien: die Erfüllung ist nicht unbedingt notwendig, sollten nur angestrebt werden, falls noch ausreichend Kapazitäten vorhanden sind.
    4. Abgrenzungskriterien: diese Kriterien sollen bewusst nicht erreicht werden
2. Produkteinsatz
    1. Anwendungsbereiche
    2. Zielgruppen
    3. Betriebsbedingungen: physikalische Umgebung des Systems, tägliche Betriebszeit, ständige Beobachtung des Systems durch Bediener oder unbeaufsichtigter Betrieb
3. Produktübersicht: kurze Übersicht über das Produkt
4. Produktfunktionen bzw. Projektumsetzung: genaue und detaillierte Beschreibung der einzelnen Produktfunktionen
5. Produktdaten: langfristig zu speichernde Daten aus Benutzersicht
6. Produktleistungen: Anforderungen bezüglich Zeit und Genauigkeit
7. Qualitätsanforderungen
8. Benutzungsoberfläche: grundlegende Anforderungen, Zugriffsrechte

9. Nichtfunktionale Anforderungen: einzuhaltende Gesetze und Normen, Sicherheitsanforderungen, Plattformabhängigkeiten
10. Technische Produktumgebung
    1. Software: für Server und Client, falls vorhanden
    2. Hardware: für Server und Client getrennt
    3. Orgware: organisatorische Rahmenbedingungen
    4. Produktschnittstellen
11. Spezielle Anforderungen an die Entwicklungsumgebung
    1. Software
    2. Hardware
    3. Orgware
    4. Entwicklungsschnittstellen
12. Gliederung in Teilprodukte
13. Ergänzungen
14. Glossar: In dem eventuelle Fachausdrücke für Laien erläutert werden.

Ganzheitliches Pflichtenheft

Ein ganzheitliches Pflichtenheft besteht nicht nur aus technischen, sondern auch aus marktwirtschaftlichen und ökologischen Anforderungen. Dabei sind Herstellung, Gebrauch und Beseitigung des Produktes einzubeziehen, mit den Auswirkungen auf Boden, Wasser und Luft. Diese Checkpoints sind übersichtlich im Bilanzierungswürfel dargestellt, der bei der Kontrolle auf Vollständigkeit des ganzheitlichen Pflichtenheftes hilft.

Siehe auch

- Anforderungserhebung
- Anforderungsmanagement (Requirementsmanagement)

Weblinks

- IT-Projektmanagement - Begriffsabgrenzung Pflichtenheft und DV-Konzept [2]
- Checklisten für Pflichtenhefte aus dem IT-Bereich [3]
- Pflichtenheft-Beispiel [4]
- Braunbuchbeschreibungen vom Institut für Rundfunktechnik IRT [5]

Einzelnachweise

[1] nach Helmut Balzert
[2] http://www.philipphauer.de/study/pm/it-projektmanagement.php#voranalyse
[3] http://www.checkliste.de/informationstechnik/anforderungsprofile-pflichtenhefte-lastenhefte/
[4] http://www.stefan-baur.de/cs.se.pflichtenheft.html
[5] http://www.irt.de/IRT/publikationen/braunbuch.htm

Software Requirements Specification

Definitionen von IEEE
• SQAP – Software Quality Assurance Plan IEEE 730 • SCMP – Software Configuration Management Plan IEEE 828 • STD – Software Test Documentation IEEE 829 • SRS – Software Requirements Specification IEEE 830 • SVVP – Software Validation & Verification Plan IEEE 1012 • SDD – Software Design Description IEEE 1016 • SPMP – Software Project Management Plan IEEE 1058

Die **Software Requirements Specification** (SRS) ist ein von IEEE (Institute of Electrical and Electronic Engineers) erstmals unter (ANSI/IEEE Std 830-1984) veröffentlichter Standard zur Spezifikation von Software. Das IEEE hat die Spezifikation mehrmals überarbeitet und die momentan neueste Version ist Std 830-1998.

Die SRS umfasst das Lastenheft wie auch das Pflichtenheft.

Qualität

Die IEEE Kap. 4.3 definiert 8 Charakteristika guter SRS:

- Korrekt
- Unmehrdeutig
- Vollständig
- Konsistent
- Bewertet nach Wichtigkeit und Stabilität
- Verifizierbar
- Modifizierbar
- Verfolgbar (Tracebar)

Korrekt und Vollständig bezieht sich dabei auf die SRS bezüglich der tatsächlichen Anforderungen (externer Bezug). Konsistenz bezieht sich auf die Anforderungen in Form der SRS alleine (interner Bezug). Unmehrdeutigkeit lässt genau eine Interpretation zu, Verifizierbarkeit begrenzt die Komplexität einer Anforderungsbeschreibung zusätzlich auf ein effizient prüfbares Maß. Modifizierbarkeit setzt insbesondere Redundanzfreiheit voraus. Traceability umfasst die vor- und rückwärtige Richtung.

Dokumentation

Die IEEE hat mit dieser Definition festgelegt, wie das Dokument aufgebaut werden soll. Die Kapitel, die in diesem Dokument vorkommen sollen, stehen somit fest. Dabei ist das Dokument grundsätzlich in 2 Bereiche aufgeteilt:

- C-Requirement (Customer-Requirement): Bereich ist mit Lastenheft vergleichbar
- D-Requirement (Development-Requirement): Bereich ist mit Pflichtenheft vergleichbar

Unter C-Requirement sind die Anforderungen aus Sicht des Kunden und/oder des End-Anwenders zu erfassen. Unter D-Requirement versteht man die Entwicklungs-Anforderungen. Dies ist die Sicht aus den Augen des Entwicklers, der technische Aspekte in den Vordergrund stellt, im Gegensatz zum Kunden.

Mit *Requirements* (deutsch: *Anforderungen*) ist sowohl die qualitative als auch die quantitative Definition eines benötigten Programms aus der Sicht des Auftraggebers gemeint. Im Idealfall umfasst eine solche Spezifikation ausführliche Beschreibung von Zweck, geplantem Einsatz in der Praxis sowie dem geforderten Funktionsumfang einer Software.

Hierbei sollte fachlichen - „Was soll die Software können?“ - wie auch technischen Aspekten - „In welchem Umfang und unter welchen Bedingungen wird die Software eingesetzt werden?“ - Rechnung getragen werden.

Eine SRS enthält nach IEEE Standard mindestens drei Hauptkapitel. Die vorgeschlagene Gliederung sollte zwar in den Kernpunkten eingehalten werden. In der Praxis wird diese jedoch häufig im Detail modifiziert. Eine exemplarische Gliederung könnte wie folgt aussehen:

- Name des Softwareprodukts
- Name des Herstellers
- Versionsdatum des Dokuments und / oder der Software

1. Einleitung
    1. Zweck (des Dokuments)
    2. Umfang (des Softwareprodukts)
    3. Verweise auf sonstige Ressourcen oder Quellen
    4. Erläuterungen zu Begriffen und / oder Abkürzungen
    5. Übersicht (Wie ist das Dokument aufgebaut?)
2. Allgemeine Beschreibung (des Softwareprodukts)
    1. Produktperspektive (zu anderen Softwareprodukten)
    2. Produktfunktionen (eine Zusammenfassung und Übersicht)
    3. Benutzermerkmale (Informationen zu erwarteten Nutzern, z.B. Bildung, Erfahrung, Sachkenntnis)
    4. Einschränkungen (für den Entwickler)
    5. Annahmen und Abhängigkeiten (nicht Realisierbares und auf spätere Versionen verschobene Eigenschaften)
3. Spezifische Anforderungen (im Gegensatz zu 2.)
    1. funktionale Anforderungen (Stark abhängig von der Art des Softwareprodukts)
    2. nicht-funktionale Anforderungen
    3. externe Schnittstellen
    4. Design Constraints
    5. Anforderungen an Performance
    6. Qualitätsanforderungen
    7. Sonstige Anforderungen

Die Schwierigkeiten, die sich in der Praxis bei einer solchen Anforderungsanalyse ergeben, sind

- mögliche Interessenkonflikte, also unterschiedliche Ziele seitens der Nutzer
- unklare oder sogar unbekannte technische Rahmenbedingungen
- sich ändernde Anforderungen oder Prioritäten schon während des Entwurfsprozesses

Literatur

- *IEEE Guide to Software Requirements Specification*, ANSI/IEEE Std 830-1984, IEEE Press, Piscataway, New Jersey, 1984.
- Andreas Kress, Robert Stevenson, Rupert Wiebel, Colin Hood, Gerhard Versteegen *Requirements Engineering Methoden und Techniken, Einführungsszenarien und Werkzeuge im Vergleich, iX Studie Anforderungsmanagement* Heise 2005 zweite Auflage 2007, ISBN 9783936931198

Weblinks

- IEEE Software Engineering Collection via the IEEE Shop [1]

Referenzen

[1] https://sbwsweb.ieee.org/ecustomercme_enu/start.swe?SWECmd=GotoView&SWEView=Catalog+View+(eSales)_Standards_IEEE&mem_type=Customer&SWEHo=sbwsweb.ieee.org&SWETS=1192713657

Objektorientierte Programmierung

Die **objektorientierte Programmierung** (kurz **OOP**) ist ein auf dem Konzept der Objektorientierung basierender Programmierstil. Die Grundidee dabei ist, Daten und Funktionen, welche auf diese Daten angewandt werden können, möglichst eng in einem sogenannten Objekt zusammenzufassen und nach außen hin zu *kapseln*, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können.

Begriffe

Im Vergleich mit anderen Programmiermethoden verwendet die objektorientierte Programmierung neue, andere Begriffe.

Die einzelnen Bausteine, aus denen ein objektorientiertes Programm während seiner Abarbeitung besteht, werden als Objekte bezeichnet. Die Konzeption dieser Objekte erfolgt dabei in der Regel auf Basis der folgenden Paradigmen:

Abstraktion

> Jedes Objekt im System kann als ein abstraktes Modell eines *Akteurs* betrachtet werden, der Aufträge erledigen, seinen Zustand berichten und ändern und mit den anderen Objekten im System kommunizieren kann, ohne offenlegen zu müssen, wie diese Fähigkeiten implementiert sind (vgl. abstrakter Datentyp (ADT)). Solche Abstraktionen sind entweder Klassen (in der klassenbasierten Objektorientierung) oder Prototypen (in der prototypbasierten Programmierung).

Klasse

> Die Datenstruktur eines Objekts wird durch die Attribute (auch *Eigenschaften*) seiner Klassendefinition festgelegt. Das Verhalten des Objekts wird von den Methoden der Klasse bestimmt. Klassen können von anderen Klassen *abgeleitet* werden (Vererbung). Dabei erbt die Klasse die Datenstruktur (*Attribute*) und die Methoden von der *vererbenden* Klasse (Basisklasse).

Prototypenbasierte ProgrammierungPrototyp Objekte werden durch das Klonen bereits existierender Objekte erzeugt und können anderen Objekten als Prototypen dienen und damit ihre eigenen Methoden zur Wiederverwendung zur Verfügung stellen, wobei die neuen Objekte nur die Unterschiede zu ihrem Prototypen-Objekt definieren müssen.

Datenkapselung

> Als Datenkapselung bezeichnet man in der Programmierung das Verbergen von Implementierungsdetails. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über definierte Schnittstellen. Objekte können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder ändern. Ein Objekt hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit dem Objekt interagiert werden kann. Dies verhindert das Umgehen von Invarianten des Programms.

Polymorphie

> Verschiedene Objekte können auf die gleiche Nachricht unterschiedlich reagieren. Wird die Zuordnung einer Nachricht zur Reaktion auf die Nachricht erst zur Laufzeit aufgelöst, dann wird dies auch späte Bindung genannt.

Feedback

Verschiedene Objekte kommunizieren über einen Nachricht-Antwort-Mechanismus, der zu Veränderungen in den Objekten führt und neue Nachrichtenaufrufe erzeugt. Dafür steht die Kopplung als Index für den Grad des Feedbacks.

Vererbung

Vererbung heißt vereinfacht, dass eine abgeleitete Klasse die Methoden und Attribute der Basisklasse ebenfalls besitzt, also „erbt“. Somit kann die abgeleitete Klasse auch darauf zugreifen. Neue Arten von Objekten können auf der Basis bereits vorhandener Objektdefinitionen festgelegt werden. Es können neue Bestandteile hinzugenommen werden oder vorhandene überlagert werden.

Persistenz

Objektvariablen existieren, solange die Objekte vorhanden sind und „verfallen“ nicht nach Abarbeitung einer Methode.

Klassen

→ *Hauptartikel:* Klasse

Zur besseren Verwaltung gleichartiger Objekte bedienen sich die meisten Programmiersprachen des Konzeptes der Klasse. Klassen sind Vorlagen, aus denen "Instanzen" genannte Objekte zur Laufzeit erzeugt werden. Im Programm werden dann nicht einzelne Objekte, sondern eine Klasse gleichartiger Objekte definiert. Existieren in der erwählten Programmiersprache keine Klassen oder werden diese explizit unterdrückt, so spricht man zur Unterscheidung oft auch von *objektbasierter* Programmierung.

Man kann sich die Erzeugung von Objekten aus einer Klasse vorstellen wie das Fertigen von Autos aus dem Konstruktionsplan eines bestimmten Fahrzeugtyps. Klassen sind die Konstruktionspläne für Objekte.

Die Klasse entspricht in etwa einem komplexen Datentyp wie in der prozeduralen Programmierung, geht aber darüber hinaus: Sie legt nicht nur die Datentypen fest, aus denen die mit Hilfe der Klassen erzeugten Objekte bestehen, sie definiert zudem die Algorithmen, die auf diesen Daten operieren. Während also zur Laufzeit eines Programms einzelne Objekte miteinander interagieren, wird das Grundmuster dieser Interaktion durch die Definition der einzelnen Klassen festgelegt.

Beispiel

Die Klasse „Auto“ legt fest, dass das Auto vier Reifen, fünf Türen, einen Motor und fünf Sitze hat.

Das Objekt „Automodell1“ hat schließlich vier Reifen mit dem Durchmesser 60 cm und der Breite 20 cm, fünf rote Türen, einen Motor mit 150 kW und fünf Ledersitze.

Ein weiteres Objekt „Automodell2“ hat vier Reifen mit dem Durchmesser 40 cm und der Breite 15 cm, fünf blaue Türen …

Beide Objekte sind unterschiedlich, gehören aber zu der gemeinsamen Klasse Auto.

Methoden

Die einer Klasse von Objekten zugeordneten Algorithmen bezeichnet man auch als Methoden.

Häufig wird der Begriff *Methode* synonym zu den Begriffen Funktion oder Prozedur aus anderen Programmiersprachen gebraucht. Die Funktion oder Prozedur ist jedoch eher als Implementierung einer Methode zu betrachten. Im täglichen Sprachgebrauch sagt man auch „*Objekt* A *ruft Methode* m *von Objekt* B *auf.“*

Eine besondere Rolle spielen Methoden für die Kapselung, insbesondere die Zugriffsfunktionen. Spezielle Methoden zur Erzeugung bzw. *Zerstörung* von Objekten heißen Konstruktoren und Destruktoren.

In vielen objektorientierten Programmiersprachen lässt sich festlegen, welche Objekte eine bestimmte Methode aufrufen dürfen. So unterscheiden beispielsweise die Programmiersprachen Java und PHP vier Zugriffsebenen, die bereits zur Übersetzungszeit geprüft werden.

1. *Private* Methoden können nur von anderen Methoden derselben Klasse aufgerufen werden.
2. Methoden auf *Paket-Ebene* können nur von Klassen aufgerufen werden, die sich im selben Paket befinden.
3. *Geschützte* (*protected*) Methoden dürfen von Klassen im selben Paket und abgeleiteten Klassen aufgerufen werden.
4. *Öffentliche* (*public*) Methoden dürfen von allen Klassen aufgerufen werden.

Analog zu diesen vier Zugriffsebenen sind in der Unified Modeling Language (UML) vier Sichtbarkeiten für Operationen definiert.

Attribute

Objekte (Fenster, Buttons, Laufleisten, Menüs, …) besitzen verschiedene Eigenschaften (Farbe, Größe, Ausrichtung, …). Diese Eigenschaften eines Objekts heißen *Attribute*.

Polymorphie

→ *Hauptartikel:* Polymorphie (Programmierung)

Unter bestimmten Voraussetzungen können Algorithmen, die auf den Schnittstellen eines bestimmten Objekttyps operieren, auch mit davon abgeleiteten Objekten zusammenarbeiten.

Geschieht dies so, dass durch Vererbung überschriebene Methoden an Stelle der Methoden des vererbenden Objektes ausgeführt werden, dann spricht man von Polymorphie. Polymorphie stellt damit eine Möglichkeit dar, einer durch ähnliche Objekte ausgeführten Aktion einen Namen zu geben, wobei jedes Objekt die Aktion in einer für das Objekt geeigneten Weise implementiert.

Diese Technik, das so genannte *Overriding*, implementiert aber keine universelle Polymorphie, sondern nur die sogenannte Ad-hoc-Polymorphie.

Bezeichnungen

Die Begriffe der objektorientierten Programmierung haben teilweise unterschiedliche Namen. Folgende Bezeichnungen werden synonym verwendet:

Bezeichnungen in der objektorientierten Programmierung		
Deutscher Begriff	**Alternativen**	**Englisch**
Basisklasse	Elternklasse, Oberklasse, Superklasse	superclass
Abgeleitete Klasse	Kindklasse, Unterklasse, Subklasse	subclass
Methode	Elementfunktion	method
Statische Methode	Klassenfunktion, Metafunktion	static method
Attribut	Datenelement, Eigenschaft	member
Instanz	Exemplar	instance

Objektorientierte Programmiersprachen

Objektorientierte Programmiersprachen besitzen einen speziellen Datentyp – das Objekt. Damit ermöglichen sie die Objektorientierung. Die *rein* objektorientierten Sprachen, wie Smalltalk, folgen dem Prinzip: „Alles ist ein Objekt." Auch elementare Typen wie Ganzzahlen werden dabei durch Objekte repräsentiert – selbst Klassen sind hier Objekte, die wiederum Exemplare von Metaklassen sind. Die verbreiteten objektorientierten Programmiersprachen, unter anderem C#, C++ und Java, handhaben das Objektprinzip nicht ganz so streng. Bei ihnen sind elementare Datentypen keine vollwertigen Objekte, da sie auf Methoden und Struktur verzichten müssen. Sie stellen dem Entwickler auch frei, wie stark er die Kapselung objektinterner Daten einhält.

Die erste bekannte objektorientierte Programmiersprache war Simula-67. Später wurden die Prinzipien dann in Smalltalk weiter ausgebaut. Mit dem ANSI/X3.226-1994-Standard wurde Common Lisp/CLOS zur ersten standardisierten objektorientierten Programmiersprache und mit ISO 8652:1995 wurde Ada 95 zur ersten nach dem internationalen ISO-Standard normierten objektorientierten Programmiersprache.

Gängige moderne Programmiersprachen (z. B. Python) unterstützen sowohl die OOP als auch den prozeduralen Ansatz, der in den üblichen Programmiersprachen der 1970er und 1980er Jahre wie Pascal, Fortran oder C vorherrschend war. Im Gegensatz dazu setzt Smalltalk, die älteste heute noch bedeutsame OOP-Sprache, auf kompromisslose Objektorientierung und hatte damit starken Einfluss auf die Entwicklung populärer OOP-Sprachen, ohne selber deren Verbreitung zu erreichen. Auch wenn der Durchbruch der OOP erst in den 1990ern stattfand, wurde die objektorientierte Programmierung bereits Ende der sechziger Jahre als Lösungsansatz für die Modularisierung und die Wiederverwendbarkeit von Code entwickelt.

Siehe auch:

- Geschichte der Programmiersprachen
- Liste objektorientierter Programmiersprachen

Techniken

In einigen objektorientierten Programmiersprachen wie JavaScript, NewtonScript und Self wird auf die Deklaration von Klassen gänzlich verzichtet. Stattdessen werden neue Objekte von bestehenden Objekten, den so genannten Prototypen, abgeleitet. Die Attribute und Methoden des Prototyps kommen immer dann zum Einsatz, wenn sie im abgeleiteten Objekt nicht explizit überschrieben wurden. Dies ist vor allem für die Entwicklung kleinerer Programme von Vorteil, da es einfacher und zeitsparend ist.

In manchen Programmiersprachen, beispielsweise in Objective C, gibt es zu jeder Klasse ein bestimmtes Objekt (Klassenobjekt), das die Klasse zur Laufzeit repräsentiert; dieses Klassenobjekt ist dann auch für die Erzeugung von Objekten der Klasse und den Aufruf der korrekten Methode zuständig.

Klassen werden in der Regel in Form von Klassenbibliotheken zusammengefasst, die häufig thematisch organisiert sind. So können Anwender einer objektorientierten Programmiersprache Klassenbibliotheken erwerben, die den Zugriff auf Datenbanken ermöglichen.

Es gibt inzwischen auch Verfeinerungen der objektorientierten Programmierung durch Methoden wie Entwurfsmuster (englisch *design patterns*), Design by Contract (*DBC*) und grafische Modellierungssprachen wie UML.

Einen immer höheren Stellenwert nimmt die aspektorientierte Programmierung ein, bei der Aspekte von Eigenschaften und Abhängigkeiten beschrieben werden. Erste Ansätze sind beispielsweise in Java mit J2EE oder der abstrakten Datenhaltung über Persistenzschichten sichtbar.

Grenzen der OOP

Das objektorientierte Paradigma hat Vor- und Nachteile je nach Anwendungsfeld in der Softwaretechnik oder konkreter Problemstellung.

Abbildung von Problemstellungen auf OOP-Techniken

Die OOP kann, wie auch andere Programmierparadigmen, verwendet werden, Probleme aus der realen Welt abzubilden. Als ein typisches Beispiel für Problemstellungen, die sich einer geschickten Modellierung mit OOP-Techniken entziehen, gilt das Kreis-Ellipse-Problem.

Objektorientierte Programmiersprachen und natürliche Sprachen

Objektorientierte Programmiersprachen können auch unter sprachwissenschaftlichen Aspekten mit natürlichen Sprachen verglichen werden. OO-Programmiersprachen haben ihren Fokus auf den Objekten, welche sprachlich Substantive sind. Die Verben ('Aktionen') sind sekundär, fest an Substantive gebunden ('gekapselt') und können i.a. nicht für sich allein stehen. Bei natürlichen Sprachen und z.B. prozeduralen Sprachen existieren Verben eigenständig und unabhängig von den Substantiven (Daten), z.B. als Imperativ und Funktion. Es kann argumentiert werden, dass diese sprachliche Einschränkung in einigen Anwendungsfällen zu unnötig komplizierten Beschreibungen von Problemen aus der realen Welt mit objektorientierten Sprachen führt.[1] [2]

OOP und Kontrollfluss

Häufig genannte Vorzüge des OOP Paradigma sind eine verbesserte Wartbarkeit und Wiederverwendbarkeit des statischen Quellcodes.[3] Hierzu werden jedoch die Kontrollflüsse und das dynamische Laufzeitverhaltens den Daten/Objekten im allgemeinen untergeordnet, abstrahiert und weggekapselt. Die Kontrollflüsse bilden sich nicht mehr für den Entwickler transparent direkt in den Codestrukturen ab (wie z.B. bei prozeduralen Sprachen), eine Umsetzung in dieser Hinsicht wird dem Compiler überlassen. Hardware-nähere Sprachen wie das prozedurale C oder Assembler bilden den echten Kontrollfluss und das Laufzeitverhalten transparenter ab.[4] Mit der wachsenden Bedeutung von paralleler Hardware und nebenläufigem Code wird jedoch eine bessere Kontrolle und Entwickler-Transparenz der komplexer werdenden Kontrollflüsse immer wichtiger – etwas, das schwierig mit OOP zu erreichen ist.[5] [6]

OOP und relationale Datenbanken

Ein häufig genannter Bereich, in dem OOP-Techniken als unzureichend gelten, ist die Anbindung von relationalen Datenbanken. OOP-Objekte lassen sich nicht direkt in allen Aspekten mit relationalen Datenbanken abbilden. Umgekehrt können über OOP die Stärken und Fähigkeiten von relationalen Datenbanken ebenfalls nicht vollständig ausgeschöpft werden. Die Notwendigkeit, eine Brücke zwischen diesen beiden Konzeptwelten zu schlagen, ist als object-relational impedance mismatch bekannt. Hierzu existieren viele Ansätze, beispielsweise die häufig

verwendete objektrelationale Abbildung, jedoch keine allgemeingültige Lösung ohne den einen oder anderen Nachteil.[7]

Laufzeitverhalten und Energieeffizienz

Die Effektivität des Laufzeitverhaltens von Anwendungen, die auf OOP-Techniken basieren, wird seit jeher kontrovers diskutiert. Alexander Chatzigeorgiou von der Universität Makedonien verglich die Laufzeiteffektivität und die Energieeffizienz von typischen Algorithmen (Gauß-Jordan-Algorithmus, Trapez-Integration und QuickSort) von prozeduralen Ansätzen und OOP-Techniken, implementiert als C- und C++-Software. Auf dem verwendeten ARM-Prozessor ergab sich für drei Algorithmen im Mittel eine um 48,41 % bessere Laufzeiteffektivität mit den prozeduralen C-Algorithmusvarianten. Es ergab sich außerdem eine im Mittel um 95,34 % höhere Leistungsaufnahme der C++-Varianten zu den C-Varianten.[8] Für Anwendungen auf mobilen Geräten, wie Handys oder MP3-Spielern mit begrenzten Leistungs- und Energiespeichervermögen, sind diese Unterschiede signifikant. Als Grund für den Unterschied in Effektivität und Energieeffizienz werden in dem Artikel generelle Abstraktions-Leistungseinbußen und die deutlich größere Anzahl von Zugriffen auf den Arbeitsspeicher durch OOP-Techniken genannt.

Kritik

- Luca Cardelli untersuchte für das DEC Systems Research Center die Effizienz von OOP-Ansätzen in dem Paper *Bad Engineering Properties of Object-Oriented Languages* mit den Metriken Programmablaufgeschwindigkeit (*economy of execution*), Compilegeschwindigkeit (*economy of compilation*), Entwicklungseffizienz für große und kleine Teams (*economy of small-scale development* und *economy of large-scale development*) und die Eleganz des Sprachumfangs selbst (*economy of language features*).[9]
- Richard Stallman schrieb 1995: „Hinzufügen von OOP zu Emacs ist ganz klar keine Verbesserung; ich verwendete OOP bei der Arbeit am Fenstersystem der Lisp-Maschine und ich stimme dem häufig gehörten, ‚der überlegene Weg zu programmieren' nicht zu."[10]
- Eine Studie von Potok et al. [11] zeigte keine signifikanten Produktivitätsunterschiede zwischen OOP und prozeduralen Ansätzen.
- Alexander Stepanow schlug vor, OOP beschreibt eine mathematisch-begrenzte Betrachtungsweise und nannte sie „fast einen genauso großen Schwindel wie die künstliche Intelligenz (KI)".[12]
- Edsger W. Dijkstra:

  > „… die Gesellschaft lechzt nach Schlangenöl. Natürlich hat das Schlangenöl die eindrucksvollsten Namen – sonst würde man es nicht verkaufen können – wie ‚Strukturierte Analyse und Design', ‚Software Engineering', ‚Maturity Models', ‚Management Information Systems', ‚Integrated Project Support Environments' ‚Object Orientation' und ‚Business Process Re-engineering' (die letzten drei auch bekannt als IPSE, OO and BPR)." – *EWD 1175: The strengths of the academic enterprise* [13][14]

Siehe auch

- Handle
- Reflexion (Programmierung)
- Schnittstelle (objektorientierte Programmierung)
- Prinzipien Objektorientierten Designs
- CORBA
- DCOM
- Geschichte der Programmiersprachen: Denkweise und Begriffe der Objektorientierung zeigten sich zuerst in Simula – einer Sprache für Simulationszwecke.

Literatur

- Bernhard Lahres, Gregor Raýman: *Praxisbuch Objektorientierung.* Galileo Computing, ISBN 3-89842-624-6 (Frei verfügbar auf der Verlags-Webseite [15]).
- Harold Abelson, Gerald Jay Sussman, Julie Sussman: *Structure and Interpretation of Computer Programs.* The MIT Press, ISBN 0-262-01153-0.
- Heide Balzert: *Objektorientierte Systemanalyse.* Spektrum Akademischer Verlag, Heidelberg 1996, ISBN 3-8274-0111-9.
- Grady Booch: *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, ISBN 0-8053-5340-2.
- Peter Eeles, Oliver Sims: *Building Business Objects.* John Wiley & Sons, ISBN 0-471-19176-0.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, ISBN 0-201-63361-2.
- Paul Harmon, William Morrissey: *The Object Technology Casebook. Lessons from Award-Winning Business Applications.* John Wiley & Sons, ISBN 0-471-14717-6.
- Ivar Jacobson: *Object-Oriented Software Engineering: A Use-Case-Driven Approach.* Addison-Wesley, ISBN 0-201-54435-0.
- Bertrand Meyer: *Object-Oriented Software Construction.* Prentice Hall, ISBN 0-13-629155-4.
- Bernd Oestereich: *Objektorientierte Programmierung mit der Unified Modeling Language.* Oldenbourg, ISBN 3-486-24319-5.
- James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design.* Prentice Hall, ISBN 0-13-629841-9.

Weblinks

- Praxisbuch Objektorientierung (openbook) [16]
- Objektorientiertes Programmieren in Java (openbook) [17]
- Aufgaben der OOP [1]
- Skriptum zu OOP, TU Wien (pdf) [18] (957 kB)
- Flash ActionScript OOP - Einführung in die objektorientierte Programmierung [19]
- Einleitung in die Objektorientiere Programmierung, Uni Wuppertal [20]
- Fachwissen auf ELEKTRONIKPRAXIS ONLINE [21] Objektorientierte Programmierung mit C

Einzelnachweise

[1] Yegge, Steve (30. März 2006). *Execution in the Kingdom of Nouns* (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html). steve-yegge.blogspot.com. Abgerufen am 3. Juli 2010.

[2] Boronczyk, Timothy (11. Juni 2009). *What's Wrong with OOP* (http://zaemis.blogspot.com/2009/06/whats-wrong-with-oop.html). zaemis.blogspot.com. Abgerufen am 3. Juli 2010.

[3] Ambler, Scott (1. Januar 1998). *A Realistic Look at Object-Oriented Reuse* (http://www.drdobbs.com/184415594). www.drdobbs.com. Abgerufen am 4. Juli 2010.

[4] Shelly, Asaf (22. August 2008). *Flaws of Object Oriented Modeling* (http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/). Intel® Software Network. Abgerufen am 4. Juli 2010.

[5] James, Justin (1. Oktober 2007). *Multithreading is a verb not a noun* (http://blogs.techrepublic.com.com/programming-and-development/?p=518). techrepublic.com. Abgerufen am 4. Juli 2010.

[6] Shelly, Asaf (22. August 2008). *HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions* (http://support.microsoft.com/?scid=kb;en-us;558117). support.microsoft.com. Abgerufen am 4. Juli 2010.

[7] Neward, Ted (26. Juni 2006). *The Vietnam of Computer Science* (http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx). Interoperability Happens. Abgerufen am 2. Juni 2010.

[8] Alexander Chatzigeorgiou: *Performance and power evaluation of C++ object-oriented programming in embedded processors*. (http://www.sciencedirect.com/science/article/B6V0B-47MJ59W-1/2/accc87febd5a5db8a73f924ea23ada47) In: *Information and Software Technology*. 45, Nr. 4, 2003, S. 195–201. doi: 10.1016/S0950-5849(02)00205-7 (http://dx.doi.org/10.1016/S0950-5849(02)00205-7).

[9] Luca Cardelli: *Bad Engineering Properties of Object-Oriented Languages*. (http://lucacardelli.name/Papers/BadPropertiesOfOO.html) In: ACM (Hrsg.): *ACM Comput. Surv*. 28, 1996, S. 150. doi: 10.1145/242224.242415 (http://dx.doi.org/10.1145/242224.242415). Abgerufen

am 21. April 2010.
[10] Stallman, Richard (16. Januar 1995). *Mode inheritance, cloning, hooks & OOP* (http://groups.google.com/group/comp.emacs.xemacs/browse_thread/thread/d0af257a2837640c/37f251537fafbb03?lnk=st&q="Richard+Stallman"+oop&rnum=5&hl=en#37f251537fafbb03). Google Groups Discussion. Abgerufen am 21. Juni 2008.
[11] Thomas Potok, Mladen Vouk, Andy Rindos: *Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment*. (http://www.csm.ornl.gov/~v8q/Homepage/Papers Old/spetep- printable.pdf) In: *Software – Practice and Experience*. 29, Nr. 10, 1999, S. 833–847. Abgerufen am 21. April 2010.
[12] Stepanow, Alexander. *STLport: An Interview with A. Stepanov* (http://www.stlport.org/resources/StepanovUSA.html). Abgerufen am 21. April 2010.
[13] http://www.cs.utexas.edu/users/EWD/transcriptions/EWD11xx/EWD1175.html
[14] Dijkstra, Edsger W. (9. Februar 1994). *EWD 1175: The strengths of the academic enterprise* (http://www.cs.utexas.edu/users/EWD/transcriptions/EWD11xx/EWD1175.html). Abgerufen am 21. April 2010.
[15] http://www.galileocomputing.de/openbook/oo/
[16] http://openbook.galileocomputing.de/oop/
[17] http://www.galileocomputing.de/openbook/javainsel7/javainsel_03_001.htm#mjf1c83f7d9f2913393ae95bd1dfa81bde
[18] http://www.complang.tuwien.ac.at/franz/objektorientiert/skript07-1seitig.pdf
[19] http://www.b-nm.at/objektorientierte-programmierung/
[20] http://www.math.uni-wuppertal.de/~axel/skripte/oop/oop1_1.html
[21] http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/implementierung/articles/252633/

Prinzipien Objektorientierten Designs

Prinzipien Objektorientierten Designs sind Prinzipien welche zu gutem objektorientierten Design führen sollen. Sie wurden neben anderen von Robert C. Martin, Bertrand Meyer und Barbara Liskov publiziert und propagiert. Viele Techniken der Objektorientierung wie Entwurfsmuster, Domain-Driven Design oder Dependency Injection basieren auf diesen Prinzipien objektorientierten Designs.

Für eine Gruppe dieser Prinzipien wurde von Robert C. Martin das Akronym “SOLID” geprägt. Diese Prinzipien gemeinsam angewandt führt lt. Robert C. Martin zu einer höheren Wartbarkeit und somit Lebensdauer von Software. Diese Prinzipien sind das “**S**ingle Responsibility Prinzip”, das “**O**pen-Closed Prinzip”, das “**L**iskovsches Substitutionsprinzip”, das “**I**nterface Segregation Prinzip” und das “**D**ependency Inversion Prinzip”.

Darüber hinaus gibt es noch eine Reihe weiterer mehr oder weniger bekannter Prinzipien objektorientierten Designs wie Design by Contract oder das Gesetz von Demeter. Eine Sonderstellung unter den Prinzipien objektorientierten Designs haben die sogenannten Packaging Prinzipien, da diese sich alle mit der Gruppierung von Klassen zu Packages beschäftigen.

SOLID Prinzipien

Single Responsibility Prinzip

Das **Single Responsibility Prinzip** besagt, dass jede Klasse nur eine einzige Verantwortung haben solle. Verantwortung wird hierbei als "Grund zur Änderung" definiert:

> „There should never be more than one reason for a class to change.“
>
> „Es sollte nie mehr als einen Grund dafür geben, eine Klasse zu ändern.“
>
> – Robert C. Martin: Agile Software Development: Principles, Patterns, and Practices[1]

Mehr als eine Verantwortung für eine Klasse führt zu mehreren Bereichen, in denen zukünftige Änderungen notwendig werden können. Die Wahrscheinlichkeit, dass die Klasse zu einem späteren Zeitpunkt geändert werden muss, steigt zusammen mit dem Risiko, sich bei solchen Änderungen subtile Fehler einzuhandeln. Dieses Prinzip führt in der Regel zu Klassen mit hoher Kohäsion, in denen alle Methoden einen starken gemeinsamen Bezug haben.

Open-Closed Prinzip

Das **Open-Closed Prinzip** besagt, dass Software-Einheiten (hier Module, Klassen, Methoden etc.) Erweiterungen möglich machen sollen (dafür offen sein), aber ohne dabei ihr Verhalten zu ändern (ihr Sourcecode und ihre Schnittstelle sollte sich nicht ändern). Es wurde 1988 von Bertrand Meyer folgendermaßen formuliert:

> „Modules should be both open (for extension) and closed (for modification)."
>
> „Module sollten sowohl offen (für Erweiterungen), als auch geschlossen (für Modifikationen) sein."
>
> – Bertrand Meyer: Object Oriented Software Construction[2]

Eine Erweiterung im Sinne des Open-Closed Prinzips ist beispielsweise die Vererbung. Diese verändert das vorhandene Verhalten einer Klasse nicht, erweitert sie aber um zusätzliche Funktionen oder Daten. Überschriebene Methoden verändern auch nicht das Verhalten der Basisklasse, sondern nur das der abgeleiteten Klasse. Folgt man weiters dem Liskovschen Substitutionsprinzip, verändern auch überschriebene Methoden nicht das Verhalten, sondern nur die Algorithmen.

Liskovsches Substitutionsprinzip

Das **Liskovsche Substitutionsprinzip** (LSP) oder **Ersetzbarkeitsprinzip** fordert, dass eine Instanz einer abgeleiteten Klasse sich so verhalten muss, dass jemand, der meint, ein Objekt der Basisklasse vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird, wenn es sich dabei tatsächlich um ein Objekt des Subtyps handelt. Es wurde 1993 von Barbara Liskov und Jeannette Wing formuliert.[3] In einem nachfolgenden Artikel wurde es folgendermaßen formuliert:

> „Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T."
>
> „Sei $q(x)$ eine Eigenschaft des Objektes x vom Typ T, dann sollte $q(y)$ für alle Objekte y des Typs S gelten, wo S ein Subtyp von T ist."
>
> – Barbara H. Liskov, Jeannette M. Wing: Behavioral Subtyping Using Invariants and Constraints[4]

Damit ist garantiert, dass Operationen vom Typ Superklasse, die auf ein Objekt des Typs Subklasse angewendet werden, auch korrekt ausgeführt werden. Dann lässt sich stets bedenkenlos ein Objekt vom Typ Superklasse durch ein Objekt vom Typ Subklasse ersetzen. Objektorientierte Programmiersprachen können eine Verletzung dieses Prinzips, das aufgrund der mit der Vererbung verbundenen Polymorphie auftreten kann, nicht von vornherein ausschließen. Häufig ist eine Verletzung des Prinzips nicht auf den ersten Blick offensichtlich.[5]

Interface Segregation Prinzip

Das **Interface Segregation Prinzip** dient dazu, zu große Interfaces aufzuteilen. Die Aufteilung soll gemäß der Anforderungen der Clients an die Interfaces gemacht werden - und zwar derart, dass die neuen Interfaces genau auf die Anforderungen der einzelnen Clients passen. Die Clients müssen also nur mit Interfaces agieren, die das, und nur das können, was die Clients benötigen. Das Prinzip wurde von Robert C. Martin 1996 folgendermaßen formuliert:

> „Clients should not be forced to depend upon interfaces that they do not use."
>
> „Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden."
>
> – Robert C. Martin: The Interface Segregation Principle[6]

Mit Hilfe des Interface Segregation Prinzip ist es möglich eine Software derart in entkoppelte und somit leichter refaktorisierbare Klassen aufzuteilen, dass zukünftige fachliche oder technische Anforderungen an die Software nur geringe Änderungen an der Software selbst benötigen.

Dependency Inversion Prinzip

Das **Dependency Inversion Prinzip** beschäftigt sich mit der Reduktion der Kopplung von Modulen. Es besagt, dass Abhängigkeiten immer von konkreteren Modulen niedriger Ebenen zu abstrakten Modulen höherer Ebenen gerichtet sein sollten. Es wurde von Robert C. Martin erstmals im Oktober 1994 beschrieben[7] und später folgendermaßen formuliert:

> „A. High-level modules should not depend on low level modules. Both should depend on abstractions.
> B. Abstractions should not depend upon details. Details should depend upon abstractions."
>
> „A. Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen.
> B. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen."
>
> – Robert C. Martin: The Dependency Inversion Principle[8]

Damit ist sichergestellt, dass die Abhängigkeitsbeziehungen immer in eine Richtung verlaufen, von den konkreten zu den abstrakten Modulen, von den abgeleiteten Klassen zu den Basisklassen. Damit werden die Abhängigkeiten zwischen den Modulen reduziert und insbesondere zyklische Abhängigkeiten vermieden.

Weitere Prinzipien Objektorientierten Designs

Neben der von Robert C. Martin propagierten SOLID Gruppe von Prinzipien Objektorientierten Designs sind noch folgende Prinzipien als Prinzipien Objektorientierten Designs bekannt:

Gesetz von Demeter

Das **Gesetz von Demeter** (englisch: Law of Demeter, kurz: LoD) besagt im Wesentlichen, dass Objekte nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren sollen. Dadurch soll die Kopplung in einem Softwaresystem verringert und dadurch die Wartbarkeit erhöht werden.

Das Gesetz wurde von Karl J. Lieberherr und Ian Holland 1989 im Paper *Assuring Good Style for Object-Oriented Programs* folgendermaßen beschrieben:

> „A **supplier** object to a method M is an object to which a message is set in M. The **preferred supplier** objects to method M are: The immediate parts of self or the argument objects of M or the objects which are either objects created directly in M or objects in global variables.
> Every supplier object to a method must be a preferred supplier."
>
> „Ein **Unterstützendes Objekt** der Methode M ist ein Objekt, an welches aus M Nachrichten gesendet werden. Die **Bevorzugten Unterstützenden Objekte** der Methode M sind: Das Objekt der Methode M, die Argumente von M oder die Objekte die direkt in M erzeugt werden oder Objekte in globalen Variablen.
> Jedes Unterstützende Objekt einer Methode muss ein Bevorzugtes Unterstützendes Objekt sein."
>
> – Karl J. Lieberherr, I. Holland: Assuring good style for object-oriented programs[9]

Durch die formale Spezifikation lässt sich das Gesetz von Demeter leicht als automatisch geprüfte Softwaremetrik umsetzen. Es bietet sich somit zur Früherkennung von Kopplungsproblemen an.

Design by Contract

Das **Design by contract** Prinzip, englisch für *Entwurf gemäß Vertrag*, auch *Programming by Contract* genannt, hat das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler „Verträge“ zur Verwendung von Schnittstellen, die über deren statische Definition hinausgehen zum Ziel. Entwickelt und eingeführt wurde es von Bertrand Meyer mit der Entwicklung der Programmiersprache Eiffel.

Das reibungslose Zusammenspiel der Programmmodule wird durch einen „Vertrag“ erreicht, der beispielsweise bei der Verwendung einer Methode einzuhalten ist. Dieser besteht aus

- Vorbedingungen (englisch *precondition*), also den Zusicherungen, die der Aufrufer einzuhalten hat
- Nachbedingungen (englisch *postcondition*), also den Zusicherungen, die der Aufgerufene einhalten wird, sowie den
- Invarianten (englisch *class invariants*), quasi dem Gesundheitszustand einer Klasse.

Der Einsatz des Design by Contract Prinzips führt zu sicherer und weniger fehleranfälliger Software, da eine Verletzung des Vertrages zu einem sofortigen Fehler (fail fast) bereits in der Softwareentwicklungsphase führt. Weiters kann die explizite Definition des Vertrages als Form der Dokumentation des Verhaltens des entsprechenden Moduls gesehen werden. Das wiederum führt zu einer besseren Erlernbarkeit und Wartbarkeit der Software.

Datenkapselung

Datenkapselung (engl. *encapsulation*), nach David Parnas auch bekannt als *information hiding* ist nach Bertrand Meyer ein weiteres Prinzip objektorientierten Designs.[10] Es beschreibt das Verbergen von Daten oder Informationen vor dem Zugriff von außen. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt statt dessen über definierte Schnittstellen. Die in der Objektorientierung verwendeten Zugriffsarten (private, protected, ...) sowie eine Reihe von Entwurfsmustern - beispielsweise Facade - unterstützen bei der Umsetzung der Datenkapselung.

Linguistic Modular Units Principle

Das **Linguistic Modular Units Principle** (englisch für *Prinzip linguistisch-modularer Einheiten*) besagt, dass Module durch syntaktische Einheiten der verwendeten Sprache - sei es eine Programmier-, Design- oder Spezifikationssprache - repräsentiert werden müssen. Im Falle einer Programmiersprache müssen Module durch separat voneinander kompilierbare Einheiten dieser gewählten Programmiersprache repräsentiert werden:

> „Modules must correspond to syntactic units in the language used.“
>
> „Module müssen den syntaktischen Einheiten der verwendeten Sprache entsprechen.“
>
> – Bertrand Meyer: Object Oriented Software Construction[11]

Self-Documentation principle

Das **Self-Documentation Principle** (englisch für *Selbst-Dokumentierungs Prinzip*) besagt, dass alle Informationen zu einem Modul in diesem selbst enthalten sein sollten. Damit wird gefordert, dass der Code entweder selbst für sich spricht (d.h. keine weitere Dokumentation nötig hat) beziehungsweise die technische Dokumentation möglichst nahe beim Sourcecode (beispielsweise bei der Verwendung von Javadoc) liegt. Dadurch wird einerseits gewährleistet, dass die technische Dokumentation dem Code entspricht und andererseits, dass die Komplexität des Codes so gering ist, dass keine komplexe Dokumentation nötig ist:

> „The designer of a module should strive to make all information about the module part of the module itself.“
>
> „Beim Entwurf eines Modules sollte man danach trachten, dass alle Informationen über das Modul, selbst Teil des Moduls sind.“
>
> – Bertrand Meyer: Object Oriented Software Construction[12]

Uniform Access Principle

Das **Uniform Access Principle** (englisch für *Prinzip des gleichartigen Zugriffes*) fordert, dass auf alle Services eines Modules mittels einer gleichartigen Notation zugegriffen werden kann - ohne dass dadurch preisgegeben wird, ob die Services dahinter auf Datenbestände zugreifen, oder Berechnungen durchführen. Services sollten also nach außen nicht bekanntgeben, WIE sie ihre Serviceleistung erbringen, sondern nur WAS ihre Serviceleistung ist:

> „All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation."
>
> „Alle Services eines Moduls sollten mittels einer gleichartigen Notation angeboten werden; eine Notation, welche nicht preisgibt, ob sie über Datenzugriffe oder Berechnungen umgesetzt wurden."
>
> – Bertrand Meyer: Object Oriented Software Construction[13]

Single Choice Principle

Das **Single Choice Principle** besagt dass verschiedene Alternativen (beispielsweise von Daten oder Algorithmen) in einem Softwaresystem in nur einem einzigen Modul abgebildet werden sollten. Beispielsweise unterstützt das Entwurfsmuster Abstrakte Fabrik dieses Prinzip indem in der Fabrik einmal aus einem Set an Alternativen entschieden wird, welches Klassen und welche Algorithmen zu verwenden sind. Diese Entscheidung muss an keiner weiteren Stelle im Programm mehr getroffen werden:

> „Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list."
>
> „Immer wenn eine Software unterschiedliche Alternatven unterstützt, sollte es nur genau ein Modul geben, welche die Liste der Alternativen kennt."
>
> – Bertrand Meyer: Object Oriented Software Construction[14]

Persistence Closure Principle

Das **Persistence Closure Principle** besagt, dass Persistenzmechanismen Objekte mit all ihren Abhängigkeiten speichern und auch wieder laden müssen. Damit ist sichergestellt das Objekte ihre Eigenschaften durchs Speichern und darauffolgende Laden nicht verändern.

> „Whenever a storage mechanism stores an object, it must store with it the dependents of that object. Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object that has not yet been retrieved."
>
> „Wenn ein Speichermechanismus ein Objekt speichert, muss es auch dessen Abhängigkeiten speichern. Wenn ein Lademechanismus ein vorher gespeichertes Objekt lädt, muss er auch dessen bisher noch nicht geladenen Abhängigkeiten laden."
>
> – Bertrand Meyer: Object Oriented Software Construction[15]

Packaging Prinzipien

Die folgenden von Bertrand Meyer und Robert C. Martin definierten Prinzipien beschäftigen sich mit der Frage, wie man Klassen zu Modulen (Packages) zusammenführen sollte. Sie führen insbesondere zu einer hohen Kohäsion innerhalb der Module und geringen Kopplung zwischen den Modulen.

Reuse Release Equivalence Prinzip

Das **Reuse Release Equivalence Prinzip** besagt dass diejenigen Klassen, welche üblicherweise gemeinsam released werden, zu Modulen zusammengeführt werden sollten. Damit wird gewährleistet, dass die Anzahl der zu verteilenden Module möglichst gering gehalten wird, was wiederum die Aufwände der Softwareverteilung reduziert.

> „The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package."
>
> „Die Granularität der Wiederverwendung ist die Granularität der Release. Nur Komponenten, welche durch ein Tracking System released werden, können effektiv wiederverwendet werden. Diese Granularität ist das Package."
>
> – Robert C. Martin: Granularity[16]

Common Closure Prinzip

Das **Common Closure Prinzip** fordert dass alle Klassen in einem Modul gemäß dem Open-Closed Prinzip geschlossen gegenüber derselben Art von Veränderungen sein sollten. Änderungen an den Anforderungen einer Software, welche Änderungen an einer Klasse eines Moduls benötigen, sollten auch die anderen Klassen des Moduls betreffen. Die Einhaltung dieses Prinzips ermöglicht es die Software derart in Module zu zerlegen, dass (zukünftige) Änderungen in nur wenigen Modulen umgesetzt werden können. Damit ist sichergestellt, dass Änderungen an der Software ohne große Seiteneffekte und somit relativ kostengünstig gemacht werden können.

> „The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package."
>
> „Klassen eines Packages sollten gemeinsam geschlossen gegenüber denselben Arten von Veränderung sein. Eine Änderung, die ein Package betrifft, sollte alle Klassen dieses Packages betreffen."
>
> – Robert C. Martin: Granularity[17]

Common Reuse Prinzip

Das **Common Reuse Prinzip** beschäftigt sich mit der Verwendung von Klassen. Es besagt, dass diejenigen Klassen, welche gemeinsam verwendet werden, auch gemeinsam zu einem Modul zusammengefasst werden sollten. Durch die Einhaltung dieses Prinzips wird eine Unterteilung der Software in fachlich bzw. technisch zusammengehörende Einheiten sichergestellt.

> „The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all."
>
> „Die Klassen eines Packages sollten gemeinsam wiederverwendet werden. Wenn man eine Klasse eines Packages wiederverwendet, sollte man alle Klassen des Packages wiederverwenden."
>
> – Robert C. Martin: Granularity[18]

Acyclic Dependencies Prinzip

Das **Acyclic Dependencies Prinzip** fordert, dass die Abhängigkeiten zwischen Modulen zyklenfrei sein müssen. D.h. wenn Klassen in einem Modul von anderen Klassen in einem anderen Modul beispielsweise durch Vererbungs- oder Relationsbeziehungen abhängen, dann dürfen keine Klassen des anderen Moduls direkt oder indirekt von Klassen des ersteren Moduls abhängen.

> „The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependency structure."
>
> „Die Abhängigkeiten zwischen Packages müssen einem direkten azyklischen Graphen entsprechen. Das bedeutet, dass es keine Zyklen in der Abhängigkeitsstruktur geben darf."
>
> – Robert C. Martin: Granularity[19]

Derartige Zyklen kann man immer aufbrechen. Dafür gibt es prinzipiell zwei Möglichkeiten:

- Anwendung des Dependency Inversion Prinzips: Wenn die Abhängigkeit von Package A auf Package B invertiert werden soll, so führe im Package von A Interfaces ein, welches die von B benötigten Methoden hält. Implementiere diese Interfaces in den entsprechenden Klassen von Package B. Somit wurde die Abhängigkeit zwischen Package A und B invertiert.
- Restrukturierung der Packages: Sammle alle Klassen eines Zyklus in einem eigenen Package zusammen und führe ein oder mehrere neue Packages ein, befüllt mit den Klassen von denen die Klassen außerhalb des Zyklus abhängen

Stable Dependencies Prinzip

Das **Stable Dependencies Prinzip** besagt, dass die Abhängigkeiten zwischen Modulen in Richtung der größteren Stabilität der Module gerichtet sein sollten. Ein Modul sollte also nur von Modulen abhängig sein, welche stabiler sind als es selbst.

> „The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is."
>
> „Die Abhängigkeiten zwischen Packages sollten in der selben Richtung wie die Stabilität verlaufen. Ein Package sollte nur von Packages abhängen, welche stabiler als es selbst sind."
>
> – Robert C. Martin: Stability[20]

Unter Stabilität versteht man hier das umgekehrte Verhältnis der ausgehenden Abhängigkeiten zur Summe aller Abhängigkeiten. Je weniger Abhängigkeiten ein Modul zu anderen hat und je mehr Abhängigkeiten andere Module zu diesem Modul haben, umso stabiler ist das Modul. Die Stabilität berechnet sich indirekt über die Instabilität folgendermaßen:

$$I = \frac{Aa}{(Ae + Aa)}$$

I .. Instabilität eines Moduls
Ae .. eingehende Abhängigkeiten (engl. *afferent couplings*).
Aa .. ausgehende Abhängigkeiten (engl. *efferent couplings*).

wobei sich Ae und Aa folgendermaßen berechnen:

- Ae = Klassen außerhalb des Moduls, welche von Klassen innerhalb des Moduls abhängen
- Aa = Klassen des Moduls, welche von Klassen außerhalb des Moduls abhängen

Die Instabilität liegt im Bereich von 0 bis 1, eine Instabilität von 0 weist auf ein maximal stabiles Modul hin, eine von 1 auf ein maximal instabiles Modul.

Stable Abstractions Principle

Das **Stable Abstractions Prinzip** fordert, dass die Abstraktheit eines Moduls direkt proportional zu seiner Stabilität sein muss.

> „Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability."
>
> „Packages die maximal stabil sind sollten maximal abstrakt sein. Instabile Packages sollten konkret sein. Die Abstraktheit eines Packages sollte proportional zu seiner Stabilität sein."
>
> – Robert C. Martin: Stability[21]

Je abstrakter ein Modul ist - d.h. je mehr Interfaces, abstrakte Klassen und Methoden es hat, desto stabiler sollte es sein. Generell errechnet sich die Abstraktheit eines Moduls folgendermaßen:

$$A = \frac{Ka}{K}$$

A .. Abstraktheit eines Moduls
Ka .. Anzahl abstrakter Klassen eines Moduls
K .. Gesamtanzahl der Klassen eines Moduls

Für jedes Modul lässt sich die Distanz zur idealen Linie - engl. Main Sequence genannt - zwischen maximaler Stabilität und Abstraktheit und maximaler Instabilität und Konkretheit errechnen. Diese reicht von 0 bis ~0,707:

$$D = \frac{A + I - 1}{\sqrt{2}}$$

D .. Distanz zur idealen Linie
A .. Abstraktheit eines Moduls
I .. Instabilität eines Moduls

Je größter die Distanz ist, desto schlechter ist das Stable Abstractions Prinzip erfüllt.

Siehe auch

- Ockhams Rasiermesser - ein Sparsamkeitsprinzip aus der Wissenschaftstheorie
- Entwurfsmuster - lösen wiederkehrende Entwurfsprobleme basierend auf den Designprinzipien
- GRASP - Entwurfsmuster, mit denen die Zuständigkeit bestimmter Klassen objektorientierter Systeme festgelegt wird
- Domain-Driven Design - Vorgehensmodell zur Modellierung komplexer Softwaresysteme
- Software-Prinzipien

Literatur

- Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558.
- Richard Pawson, Robert Matthews: *Naked Objects.* John Wiley & Sons, 30. Oktober 2002, ISBN 978-0470844205.
- Robert C. Martin: *Design Principles and Design Patterns.* 2000 (http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf).
- Robert C. Martin: *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 25. Oktober 2002, ISBN 978-0135974445.

Weblinks

- http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign
- http://c2.com/cgi/wiki?OoDesignPrinciples

Einzelnachweise

[1] Robert C. Martin: *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 25. Oktober 2002, ISBN 978-0135974445, S. 149-153 (http://www.objectmentor.com/resources/articles/srp.pdf).

[2] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 57-61.

[3] Barbara H. Liskov, Jeannette M. Wing: *Family Values: A Behavioral Notion of Subtyping*. Pittsburgh 1993.

[4] Barbara H. Liskov, Jeannette M. Wing: *Behavioral Subtyping Using Invariants and Constraints.* In: MIT Lab. for Computer Science, School of Computer Science, Carnegie Mellon University (Hrsg.): *{{{Sammelwerk}}}*. Prentice Hall, Pittsburgh Juli 1999 (http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps).

[5] Lahres , Rayman: *Praxisbuch Objektorientierung*. Seiten 153–189, siehe Literatur

[6] Robert C. Martin: *The Interface Segregation Principle.* Object Mentor, 1996 (http://www.objectmentor.com/resources/articles/isp.pdf).

[7] Robert C. Martin: *Object Oriented Design Quality Metrics.* an analysis of dependencies. In: *C++ Report.* September/Oktober 1995 Auflage. 28. Oktober 1994 (http://www.objectmentor.com/resources/articles/oodmetrc.pdf, abgerufen am 26. Juli 2010).

[8] Robert C. Martin: *The Dependency Inversion Principle.* bject Mentor, Mai 1996 (http://www.objectmentor.com/resources/articles/dip.pdf).

[9] Karl J. Lieberherr, I. Holland: *Assuring good style for object-oriented programs.* In: *IEEE Software.* S. 38 - 48 (http://citeseer.ist.psu.edu/lieberherr89assuring.html, abgerufen am 27. Februar 2010).

[10] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 18.

[11] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 53-54.

[12] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 54-55.

[13] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 55-57.

[14] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 61-63.

[15] Bertrand Meyer: *Object Oriented Software Construction.* Prentice Hall, 1988, ISBN 978-0136291558, S. 252-253.

[16] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1996, S. 4 (http://www.objectmentor.com/resources/articles/granularity.pdf, abgerufen am 24. April 2010).

[17] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1996, S. 6 (http://www.objectmentor.com/resources/articles/granularity.pdf, abgerufen am 24. April 2010).

[18] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1996, S. 5 (http://www.objectmentor.com/resources/articles/granularity.pdf, abgerufen am 24. April 2010).

[19] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1996, S. 6 (http://www.objectmentor.com/resources/articles/granularity.pdf, abgerufen am 24. April 2010).

[20] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1997, S. 8-11 (http://www.objectmentor.com/resources/articles/stability.pdf, abgerufen am 24. April 2010).

[21] Robert C. Martin: *Granularity.* In: *IEEE Software.* Dezember 1997, S. 11-14 (http://www.objectmentor.com/resources/articles/stability.pdf, abgerufen am 24. April 2010).

Quelle(n) und Bearbeiter des/der Artikel(s)

Objektorientierte Analyse und Design *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77296515 *Bearbeiter*: ABF, Chiccodoro, Druffeler, Erud, Gaius L., Info-Partikel, Javaaner, Kaisersoft, NoComment, Peter200, Sebastian.Dietrich, Smeiko, Succu, 12 anonyme Bearbeitungen

Entity-Relationship-Modell *Quelle*: http://de.wikipedia.org/w/index.php?oldid=80038446 *Bearbeiter*: Achim Raschka, Aka, Alexander.stohr, Alfred Grudszus, Andorna, Androl, Avron, Ben Ben, Bense, Bernd vdB, Bernhard Wallisch, Blubbalutsch, Cami de Son Duc, Chaosdeckel, Christian Storm, Colognese, Complex, Conny, D, D235, Der Hakawati, Der fette mo, DerHexer, Diba, Duesentrieb, Edoe, Eisenberg, Empro2, Erzbischof, Euphoriceyes, Finrod, Fleshgrinder, Frank Roeing, Freedomsaver, Friedemann Lindenthal, Geof, Gerbil, Gerold Broser, Gnu1742, Guandalug, Gurumaker, Hardenacke, He3nry, Head, Herr Th., Howwi, JakobVoss, Jengelh, Jochen, JohnTB, Kallistratos, Koerpertraining, KraetziChriZ, Krawi, Kubrick, Kurt Jansson, LKD, Laluenne, Liberal Freemason, Logograph, MAK, MFM, MaZder, Magnus, Maqqusz, MarkusHagenlocher, Matze12, MauriceKA, MichaelDiederich, Michail, MiersA, Mk85, Mm pedia, Mrieken, MsChaos, Nephiliskos, Netzmeister, Obstriegel, OecherAlemanne, Oemmler, Orcus, P. Birken, Peter200, Pfalzfrank, Rat, Revvar, Robb der Physiker, RobertRoggenbuck, S.K., STBR, Schoschi, Schusch, Sebastian.Dietrich, Semper, Sgeureka, Sicherlich, Sinn, Small Axe, Sparti, Stahlkocher, Staro1, Stefan, Stern, Thetawave, TigerDE2, Tsor, Tzeh, Tönjes, Ulis, Umweltschützen, Volker Fritzsche, WIKImaniac, Wolfgang1018, Woody, YMS, YourEyesOnly, ZenoCosini, °, 282 anonyme Bearbeitungen

Klassendiagramm *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79722363 *Bearbeiter*: Abderitestatos, Abdull, Aka, Andoras, Aurora72, Aurora72de, Bassbluete, BesondereUmstaende, Claen edon, Conny, Diba, Entlinkt, Euku, Flogger, Flominator, Florian Adler, Gereon K., Gubaer, GuidoZockoll, HaSee, Haeber, Himuralibima, Jojoo, Jongleur1983, Jpp, Kadeck, Kaisersoft, Keinhaar, Khmarbaise, Korelstar, Krawi, Krd, LKD, Levin, Lkwg82, Louis Bafrance, Luebbert42, Max Vallender, MichaelS, Mikue, Mps, MrKnister, MrTux, Nothere, Nrainer, OliD, Olliy, Pelz, PerfektesChaos, Perlentaucher, Pilawa, Pittimann, Polluks, Pwjg, RalfG., Ramtam, Rdb, RohMusik, S.K., Sae1962, Seppstein, Small Axe, Stylor, Sven Jähnichen, TheWolf, Thecriz, Timk70, Trustable, Uncopy, Wiki-observer, Wolfgang1018, 93 anonyme Bearbeitungen

Pseudocode *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79442092 *Bearbeiter*: $traight-$hoota, Aka, AlfonsGeser, Andre Engels, Cocyhok, Complex, Deki, DerHexer, Gstueb, HHK, INM, Jkrieger, Joey-das-WBF, Jonathan Hornung, Kh555, Kinley, Kku, MFM, MH, OecherAlemanne, Peter200, Piecestory, Pz6j89, Rbb, Simon04, Sparti, Sprezzatura, Thornard, Tkarcher, Trustable, Tsor, Umweltschützen, 50 anonyme Bearbeitungen

Zustandsautomat *Quelle*: http://de.wikipedia.org/w/index.php?oldid=10885431 *Bearbeiter*: APPER, Abdull, Ahartmann, Aka, Albin, Alex Anlicker, Anneke Wolf, ArachanoxReal, Arbol01, Babakus, Blah, Bmr, Calle2003, Cepheiden, Christianw, ChristophDemmer, Ckeen, Complex, Conny, D, D235, Dishayloo, Dominik Peters, ElRaki, Elian, ErikDunsing, FlorianB, Frank Jacobsen, Fristu, Fuzzy, Gubaer, Gurt, HHK, Hannes Röst, HannesH, Harro von Wuff, Head, Hermes1, Howwi, Jpp, Jschlosser, Kku, Koethnig, Korinth, Krawi, Kurt Jansson, MRB, Ma-Lik, Marc van Woerkom, Marinus81, Mh, Mþ.Æröþş, Ncnever, Nerd, Nicolas17, Omit, Peterlustig, Pinguin.tk, Pv42, Quickfix, Ra'ike, Rainer Nase, Rat, ReqEngineer, Schewek, Sinn, Stephan Herz, Stern, SvenjaWendler, Sypholux, TFW, TobiasG, Tombox2005, Trublu, Udm, VictorAnyakin, 155 anonyme Bearbeitungen

Kollaborationsdiagramm *Quelle*: http://de.wikipedia.org/w/index.php?oldid=26062588 *Bearbeiter*: Dbflush, Diba, Euku, Flominator, Gubaer, Heiko, Jpp, Krd, Magnus Manske, Ninety Mile Beach, Ramtam, Regi51, RohMusik, Sargoth, Singsangsung, Udm, Wiki-observer, 23 anonyme Bearbeitungen

Sequenzdiagramm *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78534005 *Bearbeiter*: Adlange, AleXP, Bolli1983, Centic, Christian Storm, Duesentrieb, Dundak, Fredfeuerstein, Gnu1742, Gubaer, Heiko, Jed, JoBa2282, Kataniza, Krd, Kubrick, Lunochod, Lustiger seth, Magnus Manske, MisterS, Mnh, Morjgus, Philipendula, Polluks, RJensch, Raubfisch, Rosenzweig, Staro1, Stylor, Wgd, Wiki-observer, Yahp, Ypso, 57 anonyme Bearbeitungen

Anwendungsfalldiagramm *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77541201 *Bearbeiter*: Aka, Blauerflummi, Christian Storm, Conti, Euphoriceyes, Flominator, Gubaer, Hens, Ice Wiki, Kelovy, Knoppers, Krd, Matthias Reissner, Peter Walt A., Pittimann, Ramtam, S3b123, Stefan Birkner, Stylor, Temporäres Interesse, Theghaz, Ucc, Wiki-observer, 27 anonyme Bearbeitungen

Grafische Benutzeroberfläche *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78494936 *Bearbeiter*: $traight-$hoota, A.Abdel-Rahim, AHZ, Adaxl, Aka, Akrostychon, Alexander Fischer, Alexzop, Avoided, Avron, Bastie, Baumanns, Benji, Bernard Ladenthin, Bierdimpfl, Christian Speidel, Christoph73, ChristophDemmer, CommonsDelinker, Cristof, CroMagnon, Cyphex, D, Daniel FR, Deki, Der Oberlausitzer, DerHexer, Dermartn, Don Magnifico, Du3n5cH, Dunkeltron, Duschgeldrache2, Edoe, Eike sauer, ErikDunsing, Euphoriceyes, FerdiBf, FischX, Fleasoft, Fomafix, Frank Behnsen, GDK, Galapagos999, Gerold Broser, Guidod, Haeber, Hans Koberger, Harrrharrr, Hlambert63, Ichmichi, Internetbewohner, Iwoelbern, JMGeithain, JakobVoss, Jed, Johnny Controletti, JuTa, Kdwnv, Kilian Marquardt, KingMichi, Krd, Kurt Jansson, Langec, Lefou, Leider, Leo.Math, Liberatus, Liquidat, Lirum Larum, Lustiger seth, Ma-Lik, MaVoe, Markus.oehler, MarkusHagenlocher, Media lib, Memex, Memset, Mideg, Mikenolte, Millbart, Monarch, Mononoke, Mr. Anderson, Nb, Nocturne, Numbo3, Ocrho, Oms, Palica, Pandreas, Pcgod, PeterWikileser, PhilipErdös, PhilippWeissenbacher, Phobos, Pittimann, Polluks, Rainer Bielefeld, Raphael Frey, Rax, Realdojo, Renfield, SGoebel, Schotterebene, Semper, SomnusDe, Spam, Sparti, Staro1, Stefan64, Succu, Suhadi Sadono, THOMAS, Taxiarchos228, Tcp, The-pulse, The.Modificator, TheK, ThorstenS, Tim-kellner, Timk70, Tkarcher, Toke, Tommy k13, Trustable, Tsor, UlrichAAB, Vinci, Wiegels, WikiPimpi, Wikinger77, Wimox, Würfel, XJamRastafire, YMS, Ziko, Zäh-Scharp, 112 anonyme Bearbeitungen

Systementwurf *Quelle*: http://de.wikipedia.org/w/index.php?oldid=8100344 *Bearbeiter*: Alexander.stohr, Allesmüller, Baumfreund-FFM, Bildungsbürger, Cepheiden, DanielD, Don Magnifico, F. Saerdna, Fubar, Hildegund, JCS, JFKCom, Jce, KTq, Kam Solusar, Kerbel, Kivi, Knoppen, Konzales, Linear, Mchelge, Nameless, Norro, Oreg, S.K., Schwijker, Servus, Staro1, Stefan506, Thierry Gschwind, VMzB, VerwaisterArtikel, Wiegels, Zahnstein, ZweiBein, °, 27 anonyme Bearbeitungen

Unified Modeling Language *Quelle*: http://de.wikipedia.org/w/index.php?oldid=78885236 *Bearbeiter*: APPER, Aaron Müller, Aheil, Aka, Akribix, Amtiss, An.ha, Andrea Doria, Anfuehrer, Arnomane, Avron, AxelScheithauer, Björn Klippstein, Blubbalutsch, Breeze, Cactus26, Chemiewikibm, Chiccodoro, Chris828, Christian1985, Christoph D, ChristophDemmer, Cleverboy, Commons, CommonsDelinker, Complex, Crissov, D, Der fette mo, Diba, Dieter66007, Djj, DominikusH, Drcico, Edward Montgomery Harrington, El., EldurLoki, Elwood j blues, ErikDunsing, Euku, Fab, FabianLange, Fleshgrinder, Flominator, Fomafix, Frank Jacobsen, GNosis, Gaius L., Gerhardvalentin, Gerold Broser, Gigi2345, Gnu1742, Gontsaru, GordonKlimm, Gubaer, GuidoZockoll, Gutsul, Guwac, HHK, Hadhuey, Haeber, HaraldStoerrle, Hardenacke, HbJ, He3nry, Heinte, Howwi, Hubertl, Hubi, Ivpen, JCS, JFKCom, Jae, JanRieke, Jed, JensMDD, Jfwagener, Jpp, Jschlosser, JuTa, Juesch, Kaihuener, Karl-Henner, KarstenSchulz, Kek00207, Kh80, Kku, Kmx94712, Koethnig, Krawi, Krischan111, L.M. Morgenroth, Leider, Littletux, Lkwg82, Ma-Lik, Magnus, Maikel, Marioj, MarkusBuechele79, Martin-vogel, Mathetes, Matt1971, Mdd, MetaMarph, Mgehrmann, MichaelDiederich, Mijozi, Mischka., Mkleine, Mkogler, Mm pedia, Morten Haan, Mps, Mtmaassmann, Nd, Neg, Nephelin, Nicolas G., Niemeyerstein, Nikai, Nkersten, Ochsenfrosch, Olei, Omi´s Törtchen, Ot, P. Birken, Pendulin, Perlentaucher, Peter200, Philipendula, Polluks, Progman, Quirin, RMeier, Ra'ike, Rdb, Robb der Physiker, Rosenzweig, Saehrimnir, Sbremer, Schmiddtchen, Schnark, Scooty, Sebastian.Dietrich, Seewolf, Sparti, Sprezzatura, Srbauer, Ste ba, Stefan Kühn, Stefan-sander, Sterling, Sulai, Thobach, Timekeeper, Truthpath, Tsor, Tönjes, Ulrich.fuchs, Velocet, Vogella, Wachs, Weilkiti, Wiegand, Wiki-observer, WissensDürster, Wolfgang1018, Xypron, Zeno Gantner, 344 anonyme Bearbeitungen

Pflichtenheft *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79686568 *Bearbeiter*: 1-1111, APPER, Abubiju, Aggi, Aka, Andre Riemann, Andreas 06, Andruil, Antimaterie, Avron, BJ Axel, Benatrevqre, Bernburgerin, Bernd vdB, BuC-Projekt, Christian Specht, ChristophDemmer, D, DasBee, Dberlin, Deki, Der.Traeumer, Diba, Drahreg01, Dundak, Felix Stember, Fleshgrinder, Flominator, Frasta, GNosis, Gerbil, HaeB, Haeber, Heidenodawas, Istandil, Jergen, Jpp, Kku, LKD, Luigi bosca, Lupussy, Ma-Lik, Marrrci, Methossant, Michaki, Mnh, Morten Haan, Mps, Niemeyerstein, Nina, Nocturne, Ocrho, Onee, Ot, PeterFrankfurt, Pittimann, Pmatu, Ra'ike, Regi51, RichiH, Roo1812, S1, STBR, Saehrimnir, Saenger84, Saibo, Sallynase, Schandi, Schusch, Sechmet, Sinn, Softsheep, Sparti, Spuk968, Staro1, Stefanobasta, Stern, Tambora, The-Me, Thomas Springer, Thomasxb, Tivi, Totschläger, Traxer, UlrichJ, Uwe Keim, WAH, WIKImaniac, Westiandi, WhiteCrow, Xorx, YourEyesOnly, Zahnradzacken, Zeit ist unendlich, 180 anonyme Bearbeitungen

Software Requirements Specification *Quelle*: http://de.wikipedia.org/w/index.php?oldid=77534055 *Bearbeiter*: AN, Abdull, Abubiju, Aka, Arzach, Avron, ChristophDemmer, Der Hakawati, Esteiger, Fleasoft, Ghw, Hans-Jörg Günther, Heinte, Krawi, Michaki, Patchworker, Popie, Soloturn, Sparti, Sterling, WIKImaniac, Wittkowsky, 42 anonyme Bearbeitungen

Objektorientierte Programmierung *Quelle*: http://de.wikipedia.org/w/index.php?oldid=80063454 *Bearbeiter*: Addicted, Aka, Alauda, Ani6, BLueFiSH.as, Badenserbub, Besserwisserhochdrei, Bildungsbürger, Brunft, Burghard.w.v.britzke, Busfahrer, Cactus26, Can Filip Sakrak, ChristianHujer, Cimnine, Conny, D, Dabbelju, Daka, Darkone, Deffi, Denoix, DerHexer, Diba, DocSnyder, Domybest, Druffeler, Duesentrieb, Ecki, El nappo, Euku, Euphoriceyes, Feba, Fleshgrinder, Flogger, Fomafix, Forquato, Fra Diavolo, Frank Jacobsen, Freak 1.5, GNB, Gamma, Gar kein name, Gerhardvalentin, Gnu1742, Gonzo Y., Guandalug, Gubaer, Gurt, Guwac, HHK, HaeB, Haeber, Harald Tribune, He3nry, Head, Herr Schroeder, Herr Th., Hgulf, Homer Landskirty, Howwi, Hystereser, Ikonos, JRG, Jackalope, JakobVoss, Jan Giesen, JanKG, Jivee Blau, Johang, Johannes2, JohannesMarat, Jonathan Hornung, Jonny-, Joschi71, Jpp, Kaisersoft, Karl-Henner, Kh80, Klaeren, Klxtctc, Knoerz, Konrad F., Kubrick, Kuli, Kvlado80, Kölscher Pitter, LKD, Leider, Liebeskind, Lukian, MH, Magnus, Mario, Messi, Mh, Mh26, Michaelsy, Mitteleuropäer, Mmwiki, Molily, Mps, Mtwoll, Märzhase, Nephelin, Netzki, Nikkis, Nikolaus, Nopherox, Ohno, Old toby, Pascalrossi, Pemu, PerfektesChaos, PeterVitt, Philipp Claßen, Philipp Wetzlar, Piflaser, Plantexchen, Polarlys, Polluks, Rat, Rax, Rbb, Regi51, Reinhard Kraasch, ReqEngineer, Revvar, Rho, Riyadh, RokerHRO, Roo1812, S.K., Sae1962, Sammler05, Schaengel89, Schmitther, Schoelle, Sebastian.Dietrich, Seewolf, Semper, Sinn, Snert, SonniWP, SonniWP2, Sparti, Sprachpfleger, Sprezzatura, Spuk968, Srittau, Stauba, Steffen, StepStep42, StepforStep, Supaari, SuperFLoh, Tecturon, Temporäres Interesse, Terabyte, Texon, Th.Binder, Thomas280784, Thomasv, TinoStrauss, Togs, Topias2, Tsor, Tönjes, Ulrich.fuchs, UlrichJ, Umeier, UncleOwen, Uwe Gille, Wahrer, Warp, WiESi, Wilfried Elmenreich, Wilhans, Wilske, WiseWoman, Wladislaw, YMS, Zahnradzacken, 409 anonyme Bearbeitungen

Prinzipien Objektorientierten Designs *Quelle*: http://de.wikipedia.org/w/index.php?oldid=79650165 *Bearbeiter*: Aka, Der Hakawati, GeoffreyE, Nothere, Sebastian.Dietrich, 4 anonyme Bearbeitungen

Quelle(n), Lizenz(en) und Autor(en) des Bildes

Datei:Er-diagramm.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Er-diagramm.svg *Lizenz*: unbekannt *Bearbeiter*: Der Hakawati, Grmon, MaZder, 1 anonyme Bearbeitungen

Datei:ERD Darstellungen.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:ERD_Darstellungen.png *Lizenz*: Public Domain *Bearbeiter*: Frank Roeing

Datei:Class-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Class-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Class-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Class-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Class-3.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Class-3.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Class-6.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Class-6.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:uml.classdiagram.templateclass.gif *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Uml.classdiagram.templateclass.gif *Lizenz*: Public Domain *Bearbeiter*: Andoras

Datei:Interface-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Interface-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Interface-3.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Interface-3.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Generalization-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Generalization-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Association-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Association-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer, Wiki-observer

Datei:Association-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Association-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer, Wiki-observer

Datei:Komposition_Aggregation.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Komposition_Aggregation.png *Lizenz*: Public Domain *Bearbeiter*: max limper

Datei:Klassendiagramm-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Klassendiagramm-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Example de.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Example_de.png *Lizenz*: unbekannt *Bearbeiter*: Babakus

Datei:Parser de.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Parser_de.png *Lizenz*: unbekannt *Bearbeiter*: Babakus, 1 anonyme Bearbeitungen

Datei:Door moore de.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Door_moore_de.png *Lizenz*: unbekannt *Bearbeiter*: Babakus

Datei:Door mealy example de.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Door_mealy_example_de.png *Lizenz*: unbekannt *Bearbeiter*: Babakus

Datei:Fsm logic de.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Fsm_logic_de.jpg *Lizenz*: unbekannt *Bearbeiter*: TFW

Datei:Kommunikations diagramm-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kommunikations_diagramm-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Kommunikations diagramm-3.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kommunikations_diagramm-3.png *Lizenz*: Public Domain *Bearbeiter*: Gubaer

Datei:Kommunikations diagramm-4.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kommunikations_diagramm-4.png *Lizenz*: Public Domain *Bearbeiter*: Gubaer

Datei:Kommunikations diagramm-5.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kommunikations_diagramm-5.png *Lizenz*: Public Domain *Bearbeiter*: Gubaer

Bild:Sequenz diagramm-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Bild:Sequenz diagramm-3.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-3.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Bild:Sequenzdiagramm-4a.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenzdiagramm-4a.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Kataniza

Bild:Sequenz diagramm-5.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-5.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Bild:Sequenz diagramm-6.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-6.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Bild:Sequenz diagramm-7.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-7.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Systemkontext.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Systemkontext.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Akteur_actor.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Akteur_actor.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Anwendungsfall.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Anwendungsfall.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Assoziation.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Assoziation.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Multiplizität.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Multiplizität.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Generalisierung.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Generalisierung.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Generalisierung akteur.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Generalisierung_akteur.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Include.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Include.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Extend1.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Extend1.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Extend2.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Extend2.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Anwendungsfall extend.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Anwendungsfall_extend.jpg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:Ice Wiki

Datei:Use-case-7.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Use-case-7.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Oder Zeichner: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Use-case-8.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Use-case-8.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Oder Zeichner: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Kubuntu-9.10-Widgets.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Kubuntu-9.10-Widgets.png *Lizenz*: GNU General Public License *Bearbeiter*: Canonical/KDE

Datei:Schreibtischmetapher Windows 1.x.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Schreibtischmetapher_Windows_1.x.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:Qhx

Datei:Geos C64.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Geos_C64.jpg *Lizenz*: unbekannt *Bearbeiter*: Alexander.stohr, BODACIOUS, Shape

Datei:Atari 1040STf.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Atari_1040STf.jpg *Lizenz*: unbekannt *Bearbeiter*: Liftarn, Pixel8, SanderK, StuartBrady, 1 anonyme Bearbeitungen

Datei:Amiga500 system1.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Amiga500_system1.jpg *Lizenz*: unbekannt *Bearbeiter*: Bayo, Pixel8, StuartBrady

Datei:Wfw.PNG *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Wfw.PNG *Lizenz*: unbekannt *Bearbeiter*: Desertium, Qhx

Datei:Windows 95 C Startmenü System mit allen Updates 2010-03-01.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Windows_95_C_Startmenü_System_mit_allen_Updates_2010-03-01.png *Lizenz*: unbekannt *Bearbeiter*: User:Wimox

Datei:KDE-2.0-es-es.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:KDE-2.0-es-es.png *Lizenz*: unbekannt *Bearbeiter*: KDE development team

Datei:Iain-desktop.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Iain-desktop.png *Lizenz*: unbekannt *Bearbeiter*: iain

Datei:IBook G4.jpg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:IBook_G4.jpg *Lizenz*: Creative Commons Attribution 2.0 *Bearbeiter*: Grm wnr, Nilfanion, Thuresson

Datei:Ubuntu hoary de.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Ubuntu_hoary_de.png *Lizenz*: GNU General Public License *Bearbeiter*: User:Eipa

Datei:Xubuntu-6.06-pl.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Xubuntu-6.06-pl.png *Lizenz*: unbekannt *Bearbeiter*: Emx, Faxe, Imz, Mardus, Rbuj, Sven

Datei:PCLinuxOS_captura3.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:PCLinuxOS_captura3.png *Lizenz*: Creative Commons Attribution-Sharealike 2.5 *Bearbeiter*: http://es.wikipedia.org/wiki/Usuario:VARGUX

Datei:Unr-desktop-small.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Unr-desktop-small.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Ubuntu

Datei:Windows7_GUI.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Windows7_GUI.png *Lizenz*: unbekannt *Bearbeiter*: Benutzer:Domser

Datei:OO-historie-2.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:OO-historie-2.svg *Lizenz*: GNU Free Documentation License *Bearbeiter*: User:Chris828

Datei:MetamodelHierarchy_de.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:MetamodelHierarchy_de.svg *Lizenz*: Attribution *Bearbeiter*: Jens von Pilgrim

Datei:Pin-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Pin-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Activity_2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Activity_2.png *Lizenz*: unbekannt *Bearbeiter*: Mdd, Ra'ike

Datei:Use-case-6.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Use-case-6.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Oder Zeichner: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Informationflow-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Informationflow-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Sequenz diagramm-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Sequenz_diagramm-1.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Component-2.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Component-2.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Oder Zeichner: Gubaer Original uploader was Gubaer at de.wikipedia

Datei:Stereotype-4.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Stereotype-4.png *Lizenz*: GNU Free Documentation License *Bearbeiter*: Gubaer

Datei:Verteilungsdiagramm.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Verteilungsdiagramm.png *Lizenz*: Public Domain *Bearbeiter*: JuTa, Mdd

Datei:Statemachine-1.png *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:Statemachine-1.png *Lizenz*: Public Domain *Bearbeiter*: Gubaer

Datei:UML-Diagramme.svg *Quelle*: http://de.wikipedia.org/w/index.php?title=Datei:UML-Diagramme.svg *Lizenz*: Creative Commons Attribution-Sharealike 3.0 *Bearbeiter*: User:MetaMarph

Lizenz

Wichtiger Hinweis zu den Lizenzen

Die nachfolgenden Lizenzen bezieht sich auf den Artikeltext. Im Artikel gezeigte Bilder und Grafiken können unter einer anderen Lizenz stehen sowie von Autoren erstellt worden sein, die nicht in der Autorenliste erscheinen. Durch eine noch vorhandene technische Einschränkung werden die Lizenzinformationen für Bilder und Grafiken daher nicht angezeigt. An der Behebung dieser Einschränkung wird gearbeitet.
Das PDF ist daher nur für den privaten Gebrauch bestimmt. Eine Weiterverbreitung kann eine Urheberrechtsverletzung bedeuten.

Creative Commons Attribution-ShareAlike 3.0 Unported - Deed

Diese "Commons Deed" ist lediglich eine vereinfachte Zusammenfassung des rechtsverbindlichen Lizenzvertrages (http://de.wikipedia.org/wiki/Wikipedia:Lizenzbestimmungen_Commons_Attribution-ShareAlike_3.0_Unported) in allgemeinverständlicher Sprache.
Sie dürfen:

- das Werk bzw. den Inhalt **vervielfältigen, verbreiten und öffentlich zugänglich machen**
- **Abwandlungen und Bearbeitungen** des Werkes bzw. Inhaltes anfertigen

Zu den folgenden Bedingungen:

- **Namensnennung —** Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.
- **Weitergabe unter gleichen Bedingungen —** Wenn Sie das lizenzierte Werk bzw. den lizenzierten Inhalt bearbeiten, abwandeln oder in anderer Weise erkennbar als Grundlage für eigenes Schaffen verwenden, dürfen Sie die daraufhin neu entstandenen Werke bzw. Inhalte nur unter Verwendung von Lizenzbedingungen weitergeben, die mit denen dieses Lizenzvertrages identisch, vergleichbar oder kompatibel sind.

Wobei gilt:

- **Verzichtserklärung —** Jede der vorgenannten Bedingungen kann aufgehoben werden, sofern Sie die ausdrückliche Einwilligung des Rechteinhabers dazu erhalten.
- **Sonstige Rechte —** Die Lizenz hat keinerlei Einfluss auf die folgenden Rechte:
  - Die gesetzlichen Schranken des Urheberrechts und sonstigen Befugnisse zur privaten Nutzung;
  - Das Urheberpersönlichkeitsrecht des Rechteinhabers;
  - Rechte anderer Personen, entweder am Lizenzgegenstand selber oder bezüglich seiner Verwendung, zum Beispiel Persönlichkeitsrechte abgebildeter Personen.
- **Hinweis —** Im Falle einer Verbreitung müssen Sie anderen alle Lizenzbedingungen mitteilen, die für dieses Werk gelten. Am einfachsten ist es, an entsprechender Stelle einen Link auf http://creativecommons.org/licenses/by-sa/3.0/deed.de einzubinden.

Haftungsbeschränkung
Die „Commons Deed" ist kein Lizenzvertrag. Sie ist lediglich ein Referenztext, der den zugrundeliegenden Lizenzvertrag übersichtlich und in allgemeinverständlicher Sprache, aber auch stark vereinfacht wiedergibt. Die Deed selbst entfaltet keine juristische Wirkung und erscheint im eigentlichen Lizenzvertrag nicht.

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.
The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document
>
> under the terms of the GNU Free Documentation License, Version 1.2
>
> or any later version published by the Free Software Foundation;
>
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
>
> A copy of the license is included in the section entitled
>
> "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
>
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.