

UML-Glossar

Vorwort

Die nachfolgenden Seiten waren bisher Buchbestandteil von *Projektentwicklung mit UML und Enterprise Architect* bis zur Version 10.0 ISBN-10:3-9502692-0-8. Mit dem Release von Enterprise Architect 11 wurde das Buch komplett überarbeitet und der Umfang des Buches wäre mit diesem Kapitel auf über 400 Seiten gewachsen. Um das Buch weiterhin kompakt zu halten, haben wir uns entschlossen, den UML-Grundlagenteil nur noch als PDF zur Verfügung zu stellen.

Die thematische Ausrichtung hat uns auch dazu bewogen, der neuen Auflage einen neuen Buchtitel zu geben. Der neue Titel lautet Kompendium zu Enterprise Architect von SparxSystems und wird von SparxSystems mit der ISBN 13 978-3-9503784-0-5 in Deutsch bzw. unter dem Titel Compendium of Enterprise Architect from SparxSystems mit der ISBN 978-3-9503784-1-2 in Englischer Sprache in Buchform angeboten.

Die neuen Bücher sind weiterhin als Unterlage und Leitfaden für das Training "[UML mit Enterprise Architect](#)" konzipiert, eignen sich aber auch hervorragend für das Selbststudium.

Die Bücher wurden um folgende Kapitel erweitert:

- Team Collaboration – mehrere Benutzer an einem Modell
- Transparente Versionierung für Service Orientierte Architekturen
- Das Farbenspiel des Enterprise Architects
- Element Discussions
- Model Mail
- Umfassende Dokumentation: optimale Modellstruktur

Die jeweils neusten Informationen über die aktuellste Auflage und Bezugsmöglichkeiten finden Sie [hier](#).

Aus Gründen der Lesbarkeit wurden auf gender-gerechte Formulierungen verzichtet. Natürlich richten sich die Informationen und Erklärungen in diesem Glossar an Personen beiderlei Geschlechts.

Diese Unterlagen wurden mit großer Sorgfalt erstellt und geprüft. Leider können Fehler nicht ausgeschlossen werden. Die Autoren übernehmen keine Verantwortung oder Haftung für fehlerhafte Angaben. Die Screenshots wurden größtenteils Enterprise Architect 11.1, Build 1112 entnommen, bei Verwendung anderer Builds können sich in den Abbildungen Unterschiede ergeben.

Copyright

© 2014 Sparxsystems Software GmbH Wien. Alle Rechte vorbehalten. Kein Teil dieses Werkes darf ohne schriftliche Genehmigung des Herausgebers Sparxsystems Software GmbH unter Verwendung elektronischer Systeme entfremdet, geändert oder publiziert werden. Der Inhalt steht Ihnen ausschließlich zur online-Einsicht und als Download zur lesenden Verwendung zur Verfügung.



Die Autoren



Ing. Dietmar Steinpichler war selbstständig als Systementwickler im Echtzeitbereich und bei einem Telekommunikationsunternehmen als Businessanalyst und Designer tätig. Seine Spezialthemen sind Programmiersprachenentwicklung im CTI-Bereich, Mustererkennung und Abstraktionsalgorithmen. Als technischer Projektleiter hat er mehrere Großprojekte im Team mit UML-Modellierungswerkzeugen und verteilter Architektur abgewickelt.

Seit März 2007 ist er als Trainer und Berater für Sparxsystems Software GmbH europaweit tätig, mit den Schwerpunkten Qualitätssicherung, Projektprozesse und Requirements Management.

E-Mail: dietmar.steinpichler@sparxsystems.eu



Dr. Horst Kargl beschäftigt sich seit 1998 mit objektorientierter Modellierung und Programmierung. Bevor er 2008 zu SparxSystems wechselte, war er an der TU Wien als wissenschaftlicher Mitarbeiter in der Lehre tätig und forschte in mehreren Projekten an den Themen E-Learning, Semantic Web sowie modellgetriebener Software Entwicklung. Hierzu dissertierte er und hat sich mit der automatischen Integration von Modellierungssprachen beschäftigt. Während seines PhD Studiums war er bereits als freiberuflicher Mitarbeiter bei SparxSystems tätig.

Im September 2008 wechselte er fix als Trainer und Berater zu SparxSystems Software GmbH Central Europa. Seine Schwerpunkte sind Software Architektur, Code Generierung sowie die Anpassungs- und Erweiterungsmöglichkeiten von Enterprise Architect.

E-Mail: horst.kargl@sparxsystems.eu

Inhalt

UML-Glossar	1
Vorwort	1
Copyright	1
Die Autoren	2
Inhalt	3
Einführung in UML.....	5
Dokumentation	5
Vorteile von UML	5
UML Standard	5
UML-Erweiterungen in Enterprise Architect.....	6
Geschichtliche Entwicklung von UML	6
UML Diagrammtypen	8
Diagrammeinsatz	9
Grundlagen der Verhaltensmodellierung	10
Anwendungsfalldiagramm (Use Case Diagram)	11
Akteure.....	11
Anwendungsfall.....	12
System (System Boundary)	13
Beziehungen	13
Anwendungsfallbeziehungen	13
Beschreibungen und Notizen	16
Grafische Elemente	17
Beispiel	17
Kapitelrückblick	18
Aktivitätsdiagramm (Activity Diagram)	20
Aktivität	20
Tokenkonzept für Aktivitätsdiagramme	21
Verbindungen.....	21
Verzweigungen	22
Zusammenführen.....	22
Splitting (Parallelisierung) und Synchronisation.....	23
Schachteln von Aktivitätsdiagrammen	23
Verantwortlichkeitsbereiche (Swimlanes).....	24
Asynchrone Prozesse	25
Unterbrechungsbereich.....	25
Grafische Elemente	26
Beispiel	29
Kapitelrückblick	31
Zustandsdiagramm (State Machine Diagram).....	32
Zustände (States).....	33
Zustandsübergänge (Transitions)	33
Symbole.....	34
Beispiel	34
Kapitelrückblick	36
Klassendiagramm (Class Diagram)	37
Klasse.....	37
Objekt	38
Eigenschaften (Attribute).....	39
Methoden (Operationen)	39
Beziehungen	39
Schnittstellen.....	46
Symbole.....	49

Beispiel	50
Kapitelrückblick	52
Paketdiagramm (Package Diagram)	53
Kapitelrückblick	55
Richtige Antworten: 1c, 2a	55
Interaktionsdiagramm (Interaction Diagram)	56
Sequenzdiagramm (Sequence Diagram)	56
Ausführungsfokus	56
Nachrichtenarten	56
Symbole	58
Beispiel	58
Kapitelrückblick	60
Kommunikationsdiagramm (Communication Diagram)	61
Symbole	62
Beispiel	62
Sequenzdiagramme vs. Kommunikationsdiagramme	63
Kapitelrückblick	64
Interaktionsübersichtsdiagramm (Interaction Overview Diagram)	65
Komponentendiagramm (Component Diagram)	66
Symbole	66
Beispiel	67
Verteilungsdiagramm (Deployment Diagram)	68
Symbole	68
Beispiel	69
Kapitelrückblick	70
Zeitdiagramm (Timing Diagram)	71
Kompositionsstrukturdiagramm (Composite Structure Diagram)	71
Objektdiagramm (Object Diagram)	72
Abb. 64: Beispiel Objektdiagramm (rechts) und zugehöriges KlassendiagrammKapitelrückblick	72
Weiterführende Literatur - Empfehlungen	74
Abbildungsverzeichnis	78
Index	80

Einführung in UML

UML ist eine standardisierte grafische Darstellungsform zur *Visualisierung, Spezifikation, Konstruktion* und *Dokumentation* von (Software-)Systemen. Sie bietet ein Set an standardisierten Diagrammtypen, mit denen komplexe Sachverhalte, Abläufe und Systeme einfach, übersichtlich und verständlich dargestellt werden können.

UML ist keine Vorgangsweise und auch kein Prozess, sie stellt lediglich ein „Wörterbuch“ an Symbolen zur Verfügung, von denen jedes eine definierte Bedeutung hat. Sie bietet Diagrammtypen für die objektorientierte Analyse, Design und Programmierung und gewährleistet somit einen nahtlosen Übergang von den Anforderungen an ein System bis zur fertigen Implementierung. Dabei werden die Struktur und das Verhalten des Systems dargestellt und somit Angriffspunkte für eine Optimierung der Lösung geboten.

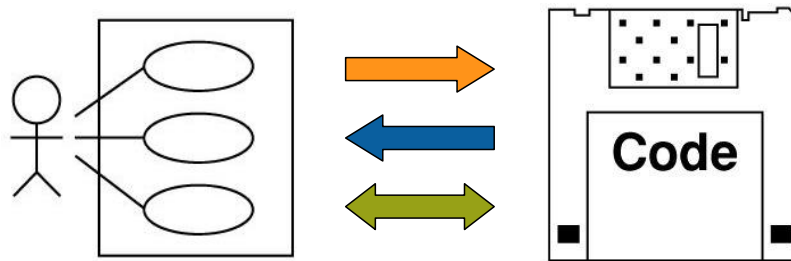


Abb. 1: Forward-, Reverse und Round-trip Engineering

Dokumentation

Ein wesentlicher Punkt von UML ist die Möglichkeit, Diagramme als Teil der Projektdokumentation verwenden zu können. Diese können in vielfältigerweise Niederschlag in den verschiedenen Dokumenten finden, beispielsweise können Use Case Diagramme zur Beschreibung der funktionalen Anforderungen in die Anforderungsdefinition wandern. Klassen- bzw. Komponentendiagramme können als Softwarearchitektur im Designdokument verwendet werden. Grundsätzlich können UML Diagramme in praktisch jeder technischen Dokumentation (z. B. Testpläne) verwendet aber auch Teil des Benutzerhandbuches werden.

Vorteile von UML

Die Verwendung von UML als "gemeinsame Sprache" führt zu einer Verbesserung der Zusammenarbeit zwischen Technikern und Nicht-Technikern, darunter fallen Projektleiter, Business Analysten, Software-/Hardwarearchitekten, -designer und -entwickler. UML hilft, Systeme besser zu verstehen, Möglichkeiten der Vereinfachung und/oder Wiederverwendbarkeit aufzudecken und mögliche Risiken besser zu erkennen. Durch frühzeitige Erkennung von Fehlern in der Analyse- bzw. Designphase eines Projekts können die Kosten während der Implementierungsphase verringert werden. Die Möglichkeit des Roundtrip Engineerings bietet vor allem für Entwickler eine enorme Zeitersparnis.

Obwohl UML ursprünglich für die Modellierung von Software-Systemen entwickelt worden ist, ist sie prinzipiell auch für Organisationsprojekte einsetzbar. Durch die Möglichkeit, Prozesse visualisieren zu können, ist es im Anschluss möglich, diese zu analysieren und zu verbessern. Auch eine Anwendung bei Softwareprojekten ohne objektorientierte Sprachen oder bei Hardwareprojekten ist möglich und sinnvoll.

UML Standard

Die offizielle Spezifikation der UML 2.4.1 ist ein komplexes, über tausend Seiten umfassendes Werk (<http://uml.org/>) und ist formal in folgende Teilspezifikationen gegliedert:

- Infrastructure (Kern der Architektur, Profiles, Stereotype),
- Superstructure (statische und dynamische Modellelemente),
- OCL (Object Constraint Language) und
- Diagram Interchange (UML-Austauschformat)

Das vorliegende Buch deckt nur die wichtigsten Kernelemente der UML ab und stellt in keiner Weise eine vollständige und umfassende Referenz dar. Für weitergehende und vertiefende Details der UML wird an weiterführende Literatur verwiesen (siehe Anhang).

UML-Erweiterungen in Enterprise Architect

Enterprise Architect nutzt den in der UML vorgesehenen Erweiterungsmechanismus (Profile) um sowohl neue Elemente – z. B. ein Element für Requirement – als auch weitere Diagrammtypen zur Verfügung zu stellen. Ebenso werden erweiterte Properties – z. B. Fenster für Tests, Arbeitsaufträge, Risiken, usw. – bereitgestellt. Dadurch entsteht ein UML-basiertes Werkzeug, das zusammen mit einer auch integrierbaren Entwicklungsplattform die umfassende Projektarbeit inklusive Requirements Management, Betriebsdokumentation, usw. erlaubt.

Geschichtliche Entwicklung von UML

Obwohl die Idee der Objektorientierung über 30 Jahre alt ist und die Entwicklung objektorientierter Programmiersprachen fast ebenso lange zurückliegt, erschienen die ersten Bücher über objektorientierte Analyse- und Designmethoden erst Anfang der 90er Jahre. Die Urväter dieser Idee waren Grady Booch, Ivar Jacobson und James Rumbaugh. Jeder dieser drei „Veteranen“ hatte seine eigene Methode entwickelt, die jedoch auf bestimmte Anwendungsbereiche spezialisiert und begrenzt war.

1995 begannen zunächst Booch und Rumbaugh, ihre Methoden in Form einer gemeinsamen Notation zur Unified Method (UM) zusammenzuführen. Die Unified Method wurde jedoch schon bald in Unified Modeling Language (UML) umbenannt, was auch eine angemessenere Bezeichnung darstellte, weil es sich im Wesentlichen nur um die Vereinheitlichung der grafischen Darstellung und Semantik der Modellierungselemente handelte und keine Methodik beschrieben wurde. Modeling Language ist im Grunde nur eine andere Umschreibung für Notation. Kurze Zeit später stieß auch Ivar Jacobson dazu, sodass die von ihm geprägten Use Cases (dt. Anwendungsfälle) integriert wurden. Die Drei nannten sich fortan „Amigos“.

Weil die Methoden von Booch, Rumbaugh und Jacobson bereits sehr populär waren und einen hohen Marktanteil hatten, bildete die Zusammenführung zur Unified Modeling Language (UML) einen Quasistandard. Schließlich wurde 1997 die UML in der Version 1.1 bei der Object Management Group (OMG) zur Standardisierung eingereicht und akzeptiert. Die Versionen 1.2 bis 1.5 enthalten jeweils einige Korrekturen. Die Version 2.0 ist seit 2004 als Standard verabschiedet worden und enthält einige wesentliche Änderungen und Erweiterungen. Die Version 2.3 wurde 2010 zum Standard, die derzeit aktuelle Version 2.4.1. im August 2011. (Quelle: OOSE)

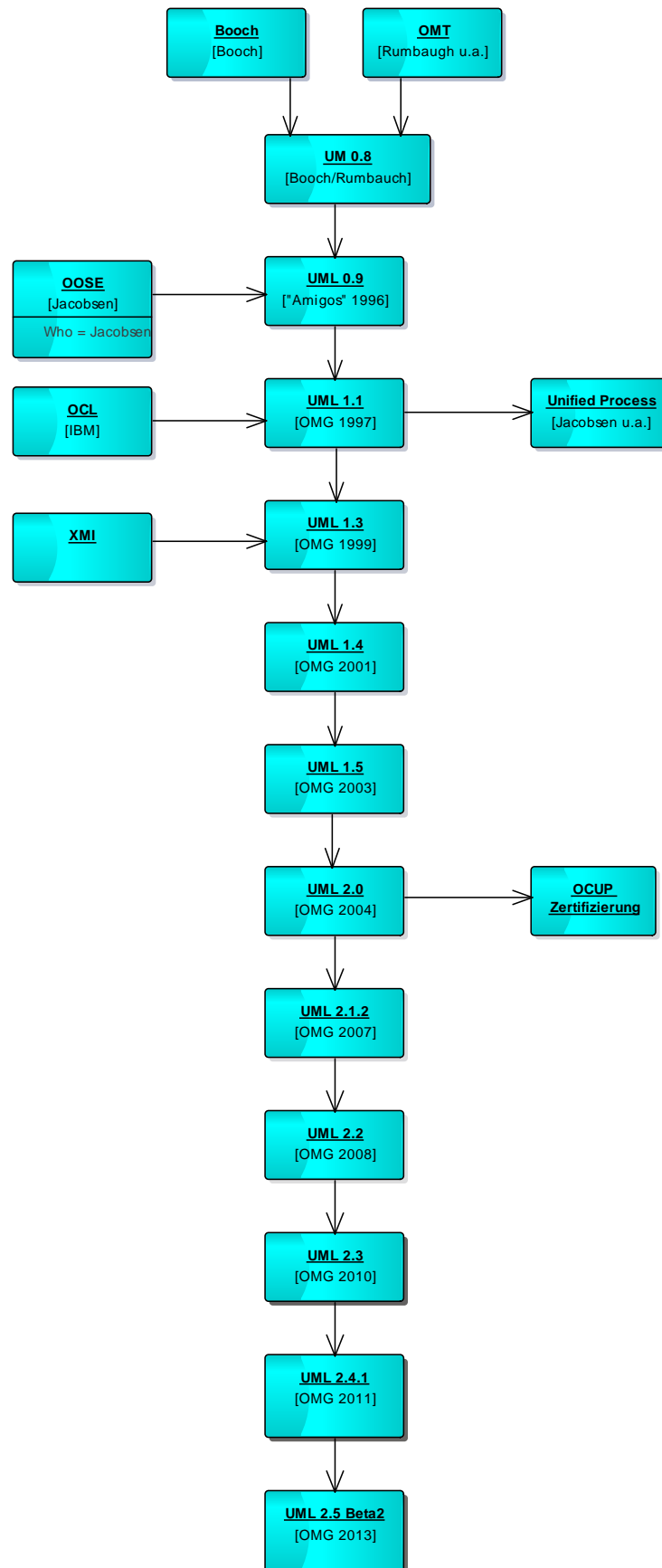


Abb. 2: Historische Entwicklung der UML

UML Diagrammtypen

Es existiert in der UML offiziell keine Diagrammübersicht oder -kategorisierung. Während UML-Modelle und das hinter den Diagrammen stehende Repository in der UML definiert sind, können Diagramme, d. h. spezielle Sichten auf das Repository, relativ frei definiert werden.

Ein Diagramm ist in der UML eigentlich mehr eine Sammlung von Notationselementen. So beschreibt beispielsweise das Paketdiagramm das Paketsymbol, die Merge-Beziehung usw. Ein Klassendiagramm beschreibt Klasse, Assoziation usw. Trotzdem dürfen natürlich Klassen und Pakete in einem Diagramm gemeinsam dargestellt werden.

Ein Diagramm besteht aus einer von einem Rechteck umgebenen Diagrammfläche und einem Diagrammkopf in der linken oberen Ecke. Im Diagrammkopf steht (optional) Diagrammtyp, (obligatorisch) Diagrammname und (optional) Parameter.

Der Diagrammtyp ist beispielsweise *sd* für Sequenzdiagramme oder *cd* für Klassendiagramme. Das Parameterfeld ist für parametrisierbare Modelle wichtig.

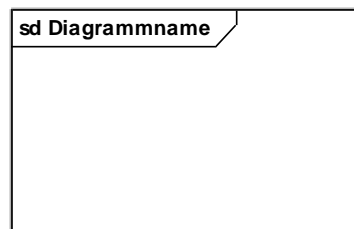


Abb. 3: Beispiel Diagrammrahmen

UML in der Version 2.4 enthält 13 Diagrammtypen, die grob in zwei Gruppen unterteilt werden können. Die Gruppe der Strukturdiagramme stellt statische Aspekte eines Systems dar, die Gruppe der Verhaltensdiagramme die dynamischen Teile.

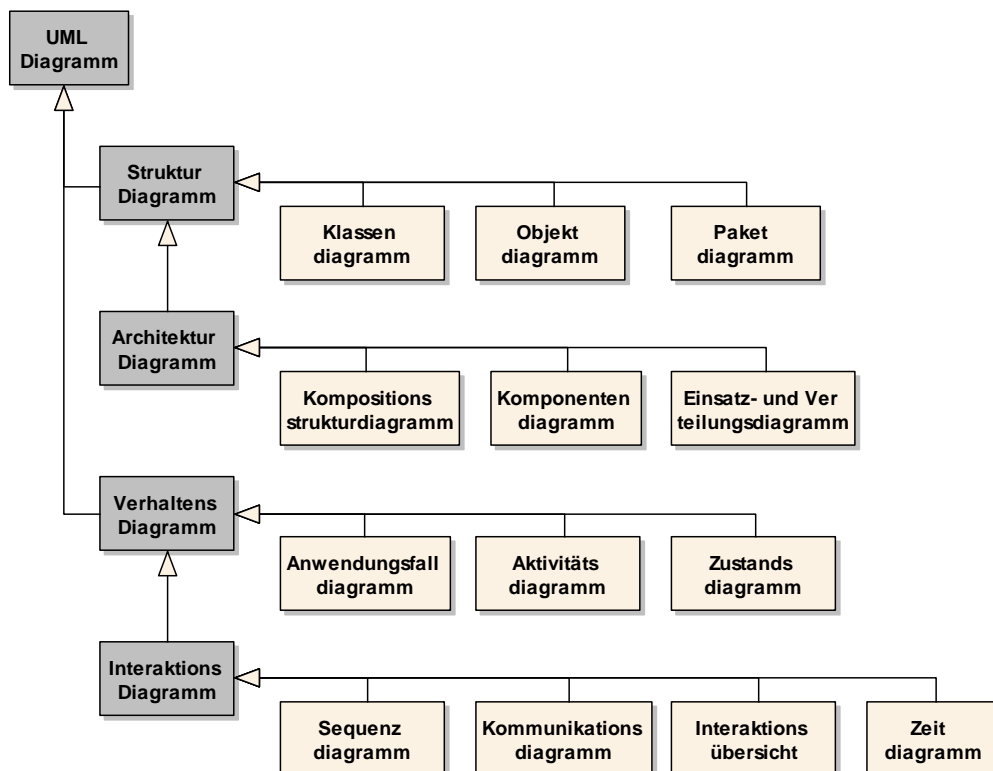


Abb. 4: UML-Diagrammarten im Überblick

Diagrammeinsatz

Vielen UML-Neulingen stellt sich bald die Frage, in welchem Zusammenhang diese Diagramme stehen. Diese Frage ist absolut berechtigt, jedoch gibt uns UML darauf keine Antwort. Erst die Vorgangsweise bei der Softwareentwicklung, also der verwendete Prozess dahinter, kann diese Frage beantworten. Ein möglicher Ansatz, in welcher Reihenfolge, also in welchen Phasen eines Projekts, die Diagramme ihren Einsatz finden, gibt die folgende Aufstellung:

Anwendungsfalldiagramm (Use Case Diagram)	Analysephase
<ul style="list-style-type: none"> o Welche Anwendungsfälle in der zu erstellenden Anwendung enthalten sind. o Welche Akteure diese Anwendungsfälle auslösen. o Welche Abhängigkeiten der Anwendungsfälle untereinander bestehen, z. B.: <ul style="list-style-type: none"> o Ob ein Anwendungsfall in einem anderen enthalten ist. o Ob ein Anwendungsfall eine Spezialisierung eines andern darstellt. o Ob ein bestehender Anwendungsfall durch einen zweiten erweitert wird. 	
Aktivitätsdiagramm (Activity Diagram)	Analyse- und Designphase
<ul style="list-style-type: none"> o Welche Schritte innerhalb eines Anwendungsfalls durchlaufen werden. o Welche Zustandsübergänge die beteiligten Objekte erfahren, wenn die Abarbeitung von einer Aktivität zur nächsten wechselt. 	
Paketdiagramm (Package Diagram)	Analyse- und Designphase
<ul style="list-style-type: none"> o In welche Pakete die Anwendung zerlegt werden kann. o Welche Pakete eine weitere Unterteilung ermöglichen. o Welche Kommunikation zwischen den Paketen realisiert werden muss. 	
Klassendiagramm (Class Diagram)	Analyse- und Designphase
<ul style="list-style-type: none"> o Welche Zusammenhänge bestehen in der Aufgabenstellung (Domainmodell). o Welche Klassen, Komponenten und Pakete beteiligt sind. o Über welche Kommunikation die Zusammenarbeit stattfindet. o Welche Methoden und Eigenschaften die Klassen benötigen. o Wie viele Objekte mindestens und höchstens in Verbindung stehen. o Welche Klassen als Container für mehrere Objekte zuständig sind. 	
Sequenzdiagramm (Sequence Diagram)	Designphase
<ul style="list-style-type: none"> o Welche Methoden für die Kommunikation zwischen ausgewählten Objekten zuständig sind. o Wie der zeitliche Ablauf von Methodenaufrufen zwischen ausgewählten Objekten stattfindet. o Welche Objekte in einer Sequenz neu erstellt und / oder zerstört werden. 	
Kommunikationsdiagramm (Communication Diagram)	Designphase
<ul style="list-style-type: none"> o Wie ausgewählte Objekte miteinander kommunizieren. o In welcher Reihenfolge die Methodenaufrufe stattfinden. o Welche alternativen Methodenaufrufe gegebenenfalls existieren. 	
Zustandsdiagramm (State Diagram)	Designphase
<ul style="list-style-type: none"> o Welche Zustandsübergänge von welchen Methodenaufrufen ausgelöst werden. o Welcher Zustand nach dem Erzeugen des Objekts eingenommen wird. o Welche Methoden das Objekt zerstören. 	
Komponentendiagramm	Designphase
<ul style="list-style-type: none"> o Wie werden Soft- und/oder Hardwareteile mit definierter Funktion und definierten Interfaces gekapselt. o Welche Komponenten haben Interfaces zueinander. o Welche Softwarteile erzeugen die Funktionalität in Komponenten. 	
Einsatz- und Verteilungsdiagramm (Deployment Diagram)	Designphase
<ul style="list-style-type: none"> o Welche PCs in der Anwendung zusammenarbeiten. o Welche Module der Anwendung auf welchem PC ausgeführt werden. o Auf welchen Kommunikationsmöglichkeiten die Zusammenarbeit basiert. 	

Die Reihenfolge der Verwendung der Diagramme kann gegebenenfalls von der in der Tabelle angegebenen abweichen, weil z. B. keine Arbeitsteilung mehrerer Programmierer zu verwalten ist. In diesem Fall kann das Paketdiagramm erst mit dem Klassendiagramm erstellt werden. Mit der Reihenfolge soll nur eine Möglichkeit gezeigt werden, wie Sie zu einem Modell ihrer Anwendung kommen und die Überleitung der Phasen gestalten können.

Ebenso wird das Anwendungsgebiet eine Auswirkung haben, eine Businessautomatisierungsaufgabe wird sich von der entstehenden Diagrammfolge her von einer Embedded- oder Realtime-Aufgabenstellung in der Reihenfolge der verwendeten Diagramme deutlich unterscheiden.

Grundlagen der Verhaltensmodellierung

Bei der Modellierung von Verhalten geht es um die Beschreibung von Abläufen, zeitlichen Abhängigkeiten, Zustandsänderungen, der Verarbeitung von Ereignissen und Ähnlichem. Die UML ist objektorientiert, d. h., Verhalten ist nichts, was eigenständig existiert, sondern es betrifft immer konkrete Objekte. Die Ausführung eines Verhaltens lässt sich im Detail immer auf ein Objekt zurückführen.

Jedes Verhalten resultiert aus Aktionen mindestens eines Objektes und führt zu Zustandsänderungen der beteiligten Objekte.

Verhalten ist in der UML grundsätzlich ereignisorientiert. Die Ausführung von Verhalten wird stets durch ein Ereignis ausgelöst. Zwei besondere Ereignisse treten immer auf: das Startereignis und das Terminierungsereignis.

Verhalten kann entweder direkt gestartet werden, dies wird Verhaltensaufrufereignis (CallBehaviorEvent) genannt, oder indirekt durch einen Trigger (TriggerEvent), beispielsweise Erreichen eines Zeitpunktes (TimeEvent), Nachrichtenempfang (ReceivingEvent) oder Erreichen bzw. Änderung eines bestimmten Wertes (ChangeEvent).

In der UML sind vier verschiedene Ausprägungen von Verhaltensbeschreibungen vorgesehen:

- Zustandsdiagramme (state machines)
- Aktivitäten und Aktionen (activities)
- Interaktionen (interaction)
- Anwendungsfälle (use cases)

Ein Anwendungsfalldiagramm ist eigentlich ein Strukturdiagramm, weil das Anwendungsfalldiagramm selbst keine Abläufe und Verhaltensweisen beschreibt, sondern nur die Struktur (Beziehungen) von Anwendungsfällen und Akteuren. In vielen Publikationen zur UML wird das Anwendungsfalldiagramm dennoch als Verhaltensdiagramm eingestuft. Sein Inhalt betrifft die gewünschte Funktionalität

Anwendungsfalldiagramm (Use Case Diagram)

Use Case Diagramme geben auf hohem Abstraktionsniveau einen sehr guten Überblick über das Gesamtsystem. Sie beschreiben die Funktionalität – die zu erbringenden Dienste und Leistungen – aus Anwendersicht. Jede Beziehung von einem Akteur (Benutzer bzw. externen Systems) zu einem Use Case führt in weiterer Folge meist zur Definition von Interaktionspunkten (Interfaces) im weiterführenden Detaildesign. Zu beachten ist, dass Anwendungsfalldiagramme selbst kein Verhalten und keine Abläufe beschreiben, sondern nur die Zusammenhänge zwischen einer Menge von Anwendungsfällen und den daran beteiligten Akteuren. Diese können zur Anforderungsanalyse und -Verwaltung verwendet werden. Ebenso wird keine Reihenfolge des Auftretens der beschriebenen Leistungen/Dienste dargestellt. Ein wesentlicher Vorteil des Use Case Diagramms liegt in der Strukturierung der funktionalen Anforderungen – was wird das System leisten können? Alle weiteren Beschreibungen können hierarchisch gegliedert, als Sub Use Cases oder durch andere Modelle, „dahinter“ aufgebaut werden. Projektabstimmung durch rasche Festbeschreibung des Aufgabenumfanges und eine darauf basierende Aufwandsabschätzung sind weitere Vorteile. Use Cases bieten somit einen Gesamtüberblick über die Funktionen des zu erstellenden Systems.

Use Cases beschreiben die Ziele der Benutzer und eignen sich daher besonders gut, um für Benutzer des Systems (Akteure) relevante funktionale Anforderungen an ein System zu analysieren. Das Use Case Diagramm besteht aus wenigen und sehr anschaulichen Elementen und ist aufgrund seiner Einfachheit bestens zur Kommunikation zwischen Auftraggeber und Auftragnehmer geeignet. Beide Seiten entwickeln ein gemeinsames Bild des Systems, so können Missverständnisse über den Funktionsumfang frühzeitig vermieden werden.

Das Use Case Diagramm ist lediglich die grafische Repräsentation von Anwendungsfällen und deren Beziehungen zur Umwelt und zueinander. Wichtige Informationen stecken in den Metainformationen eines Use Cases oder werden durch weitere Diagramme im Detail spezifiziert. Zu einem Use Case gehört mindestens: Name, Beschreibung (Notiz), Vor- und Nachbedingungen und ein Szenario mit den essenziellen Schritten, die notwendig sind, um den Anwendungsfall durchzuführen.

Durch das Sammeln der wichtigsten Informationen und Anforderungen an das System in Form von Use Cases, bietet sich der einzelne Use Case auch an, als Ausgangspunkt für einen Test Case herangezogen zu werden. Für jeden Use Case (Anwendungsfall) sollte es mindestens einen Test Case (Testfall) geben. Alle in einem Use Case definierten Vor- und Nachbedingungen (Pre- Post-Conditions), die weiteren qualitativen Anforderungen (Requirements) am Anwendungsfall sowie die einzelnen Szenarien und deren Alternativen dienen zur Ableitung der einzelnen Test Cases¹.

Das Use Case Diagramm in der nebenstehenden Abbildung zeigt zwei Anwendungsfälle und die zugehörigen Akteure. Die zwei Anwendungsfälle von oben nach unten gelesen suggerieren zwar eine Reihenfolge, diese ist aber seitens der UML weder gegeben noch vorgesehen. Das Diagramm beschreibt nur, welche Anwendungsfälle es gibt und wer daran beteiligt ist. Die Abläufe und die Reihenfolge können im Szenario² (natürlich sprachliche Beschreibung des Ablaufes eines Use Cases) oder als eigener Use Case beschrieben werden, z. B. in Aktivitätsdiagrammen, in Zustandsautomaten oder in Sequenzdiagrammen.

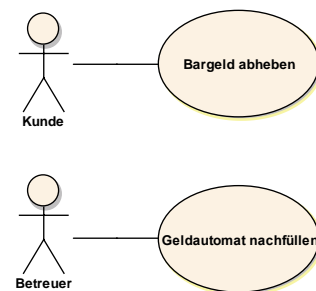


Abb. 5 Anwendungsfall

Akteure

In einem Anwendungsfalldiagramm werden alle Beteiligten (Stakeholder) eines Vorganges (Anwendungsfalles) mit Hilfe von Akteuren dargestellt. Der Akteur (*Actor*) ist definiert als eine Rolle, die sich außerhalb des Systems des zugehörigen Anwendungsfalles befindet und mit dem System, beschrieben durch seine Anwendungsfälle, interagiert. Akteure können Personen sein, die das System bedienen, oder Fremdsysteme, die auf das System zugreifen oder mit ihm interagieren.

¹ Siehe Test Cases in „Hinzufügen von Tests“ auf Seite 148

² Siehe „Bedeutung und praktische Nutzung der Eingabefelder“ auf Seite 112 ff.

ren. Sie haben Anforderungen an das oder ein Interesse am System und sind entsprechend an den Ergebnissen interessiert. Ebenso können Auslöser unabhängig von Akteuren auftreten, z. B. zeit- ablaufbedingte Trigger.

Ein Akteur beschreibt eine Rolle, die im konkreten Fall durch etwas Gegenständliches (z. B. die Person Maria Musterfrau) ersetzt werden kann. Der Akteur *Kunde* kann z. B. von jeder Person, die ein *Kunde* der *Bank* ist, ersetzt werden. Kann der Akteur nicht durch etwas Gegenständliches ersetzt werden (konkrete Person), sollte er als *Abstract* gekennzeichnet werden. Abstrakte Elemente werden in UML mit einem *kursiven* Namen geschrieben. Ein abstraktes Element kann durch keine konkrete Ausprägung in der „Wirklichkeit“ beschrieben werden, sondern dient als Abstraktion.

Die Verwendung eines Akteurs ist manchmal zu „allgemein“ und kann durch die Definition eines UML Profils verfeinert werden. Tim Weilkiens hat einen Vorschlag für ein Erweiterungsprofil im Buch „Systems Engineering mit SysML/UML“ gezeigt. Darin werden Akteure in Benutzersystem, Sensor, Aktuator, Umwelteinfluss, etc. verfeinert.

Notation von Akteuren

Die folgende Abbildung zeigt verschiedene Notationen eines Akteurs. Die UML gibt die Strichfigur als Akteur-Symbol vor. Falls kein menschlicher Akteur gemeint ist, kann alternativ das Rechteck mit Stereotyp <<actor>> verwendet werden³. Wie oben beschrieben, können auch alternative Darstellungen und Verfeinerungen des Akteurs definiert werden (Verwendung von Stereotyps). Der rechte Quader (*Node* aus dem Deployment Diagramm) kann als alternatives grafisches Symbol für ein Fremdsystem verwendet werden.



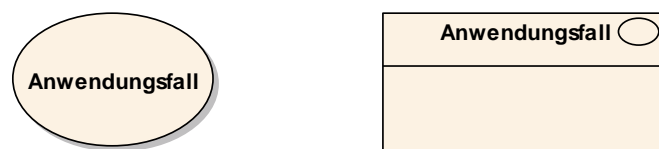
Abb. 6: Notation von Akteuren

Anwendungsfall

Ein Anwendungsfall (*Use Case*) spezifiziert eine Funktion (Menge von Aktionen), die von einem System ausgeführt werden und zu einem Ergebnis führen, das üblicherweise von Bedeutung für einen Akteur oder Stakeholder ist. Anwendungsfälle stehen für das Verhalten eines Systems und werden in der Regel durch Verhaltensdiagramme näher beschrieben. Passende Anwendungsfälle für ein Ticketsystem sind z. B. das Kaufen, das Reservieren oder das Stornieren von Eintrittskarten.

Notation von Anwendungsfällen

Die folgende Abbildung zeigt verschiedene Notationsformen von Anwendungsfällen. Die linke Abbildung ist die Standardnotation. Es ist aber auch erlaubt, den Namen des Anwendungsfalles unterhalb der Ellipse zu notieren. Das hat den Vorteil, dass die Größe der Ellipse nicht mit der Länge des Anwendungsfallnamens skalieren muss. Sowie Akteure als Rechteck mit Stereotyp Actor dargestellt werden können, ist dies auch bei Use Cases möglich. Anstelle des Stereotyps «Use Case» bietet die UML eine Darstellungsoption mit Use Case Symbol an.



³ Kontextmenü des Elements im Diagramm -> Advanced -> Use Rectangle Notation

Abb. 7: Notation von Anwendungsfällen

System (System Boundary)

Das System ist kein direktes, logisches Modellelement der UML. Mit System ist der Kontext des Anwendungsfalles gemeint, in dem die vom Anwendungsfall spezifizierten Aktionen ausgeführt werden. Das System kann dabei z. B. eine Klasse oder eine Komponente sein, welche die gesamte Anwendung repräsentiert. Das System wird durch einen oder mehrere Systemrahmen (*Boundary*) repräsentiert, die Use Cases – die Leistungen und Dienste –, die das System erbringen soll, werden in den Systemrahmen gezeichnet.

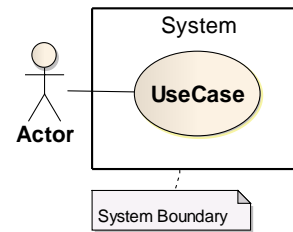


Abb. 8 System

Merke: Es ist syntaktisch falsch, Akteure innerhalb der Boundary zu zeichnen.

Beziehungen

Die Anwendungsfälle und Akteure stehen in bestimmter Beziehung zueinander. Die Beziehungen werden mit Linien modelliert. Eine solche Verbindung von Akteur und Anwendungsfall bedeutet, dass beide miteinander kommunizieren. Ein Akteur wird mittels einer einfachen Assoziation mit Anwendungsfällen verbunden. Das bedeutet in der Regel, dass der Anwendungsfall vom Akteur ausgeführt werden kann. Durch mehr Details an der Beziehung kann ein semantisch ausdrucksstärkeres Modell erstellt werden.

Wie bei Assoziationen im Klassendiagramm ist auch hier die Angabe von Multiplizitäten⁴ möglich. Die Multiplizität auf Seite des Anwendungsfalles gibt an, wie oft dieser Anwendungsfall vom Akteur gleichzeitig ausgeführt werden darf. Wenn keine Angabe gemacht wird, ist die Multiplizität immer 0..1. Auf der Seite des Akteurs bedeutet die Multiplizität, wie viele Akteure der angegebenen Rolle am Anwendungsfall beteiligt sein müssen bzw. können. Wenn keine Angabe gemacht wird, ist die Multiplizität 1..1 oder vereinfacht geschrieben 1.

Üblicherweise verwendet man keine Navigationsangaben. Gerichtete Assoziationen sind aber erlaubt. Sie bedeuten keine Datenflussrichtung – so werden sie meist interpretiert –, sondern geben den Initiator der Kommunikation zwischen Akteur und System an. Somit wird beschrieben, welcher Teil der Aktive und welcher der Passive ist. Wenn ein Akteur zu einem Anwendungsfall navigiert, dann ist der Akteur der Aktive und stößt den Anwendungsfall an. Im umgekehrten Fall, Navigation vom Anwendungsfall zum Akteur, ist der Akteur der passive und wird vom Anwendungsfall benötigt und aufgefordert teilzunehmen.

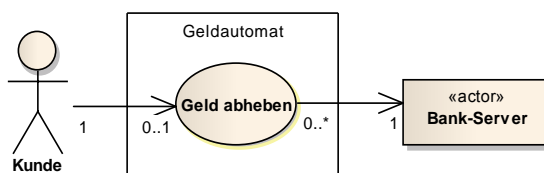


Abb. 9 Multiplizität und Aktive/Passive Akteure

Das Beispiel in Abb. 9 beschreibt, dass ein *Kunde* den Use Case *Geld abheben* anstößt, aber maximal 1x gleichzeitig. Um *Geld abheben* auszuführen, wird der Akteur *Bank-Server* benötigt (er ist passiv). Der *Bank-Server* kann allerdings in beliebig vielen *Geld abheben* Anwendungsfällen gleichzeitig involviert sein, der *Kunde* hingegen nur 1x.

Anwendungsfallbeziehungen

Anwendungsfälle können auch voneinander abhängig sein.

- Mit einer Enthält-Beziehung (*Include*) wird ein Anwendungsfall in einen anderen Anwendungsfall eingebunden und ist ein logischer Teil von diesem. Sie stellt eine zwingende Beziehung dar und wird deshalb auch oft als „Mussbeziehung“ bezeichnet.

⁴ Die Multiplizität ist ein zeitabhängiger Wert mit einer unteren und oberen Grenze, meist notiert als x..y. Beispiel: Zu einem bestimmten Zeitpunkt brauche ich 2..5 der Elemente am gegenüberliegenden Ende. Die Multiplizität beschreibt die Menge möglicher Ausprägungen, die Kardinalität hingegen eine konkrete Menge.

- Mit einer Erweiterungsbeziehung (*Extend*) hingegen lässt sich ausdrücken, dass ein Anwendungsfall unter bestimmten Umständen und an einer bestimmten Stelle (dem sog. Erweiterungspunkt, englisch *extension point*) durch einen anderen erweitert wird. Sie stellt eine optionale Beziehung dar und wird deshalb oft als „Kannbeziehung“ bezeichnet.
- Durch die Generalisierungsbeziehung (*Generalisation*) können hierarchische Zusammenhänge zwischen Anwendungsfällen beschrieben werden. Generellere Use Cases werden durch konkretere verfeinert. Ebenso können Anwendungsfälle abstrakt sein (Name ist *kursiv* geschrieben) und erst durch konkretere Anwendungsfälle „ausführbar“ werden.

Enthält-Beziehung (Include)

Teile von Anwendungsfällen, die in mehreren Use Cases in identischer Weise vorkommen, können in einem eigenen Anwendungsfall ausgelagert und per Enthält-Beziehung wieder eingebunden werden, um so eine redundante Beschreibung der identischen Teile zu vermeiden. Durch die Enthält-Beziehung werden, anders als bei der Generalisierungsbeziehung, keine Eigenschaften weitervererbt.

Die Enthält-Beziehung wird dargestellt durch einen gestrichelten Pfeil mit offener Pfeilspitze, der in Richtung des inkludierten Anwendungsfalles zeigt. Auf dem Pfeil wird das Schlüsselwort «include» notiert. Mit dem eingebundenen Anwendungsfall muss nicht zwingend ein Akteur verbunden sein.

Use Cases, die nicht direkt von einem Akteur aufgerufen werden können, werden oft mit dem Stereotyp «secondary» versehen. Das gehört nicht zum UML-Standard, es ist aber üblich Anwendungsfälle als "sekundär" zu bezeichnen, wenn sie nicht direkt von einem Akteur ausgeführt werden können, sondern nur im Kontext eines "primären" Anwendungsfalles Sinn machen oder lediglich in dessen Kontext ausführbar sein sollen. Primäre Use Cases werden nicht mit einem zusätzlichen Stereotyp gekennzeichnet!

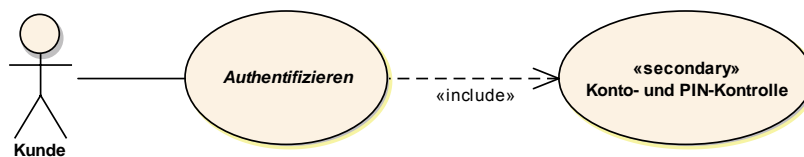


Abb. 10: Beispiel «include» Beziehung

Im Use Case Diagramm wird durch die «include» Beziehung beschrieben, dass ein Use Case IMMER einen anderen Use Case aufruft. Wann genau der eingebundene Use Case auszuführen ist, kann nicht im Diagramm beschrieben werden! Dies wird im *Use Case Szenario* textuell beschrieben oder in einem Verhaltensdiagramm, welches den Use Case detaillierter darstellt.

Hinweis: Bei der Verwendung von Enthält-Beziehungen ist darauf zu achten, dass nur Use Cases gleichen Abstraktionsniveaus verbunden werden. Das Inkludieren verleitet dazu, immer tiefer und detaillierter in das zu beschreibende System einzutauchen. Ein Use Case *PIN eingeben*, der in *Authentifizieren* enthalten sein soll (*include*), wäre zu detailliert. Hinzu kommt, dass *PIN eingeben* ein schlechter Use Case ist, da der Prozess (Workflow) hinter *PIN eingeben* zu gering ist, um einen eigenen Use Case dafür zu definieren.

Merke: Oft benötigte Sachverhalte werden als eigene Use Cases beschrieben und können durch «include» Beziehungen beliebig oft wieder verwendet werden. Jeder Use Case, der durch eine «include» Beziehungen eingebunden wurde, wird IMMER ausgeführt, wenn der einbindende Use Case ausgeführt wird!

Erweiterungsbeziehung (Extend)

Werden Teile eines Use Cases nur unter speziellen Bedingungen ausgeführt, können diese Teile als eigene Anwendungsfälle modelliert und mittels «extend» Beziehung eingebunden werden. Die Erweiterungsbeziehung zeigt auf den Anwendungsfall, der erweitert wird, und geht von dem Anwendungsfall aus, der das Verhalten der Erweiterung beschreibt (siehe Abb. 11).

Achtung: Intuitiv würde man «extend» anders herum interpretieren. Der Pfeil zeigt aber in Richtung des Use Cases, der erweitert wird (A «extend» B). Sonst müsste die Beziehung «extended by» heißen.

Der erweiterte Use Case kann optional durch einen sogenannten Erweiterungspunkt (*extension point*) genauer beschrieben werden (siehe Abb. 12). Ein Erweiterungspunkt beschreibt das Ereignis, unter dem die Erweiterung aktiviert wird. Ein Anwendungsfall kann beliebig viele Erweiterungspunkte definieren. Zusätzlich zum Erweiterungspunkt können auch noch Bedingungen definiert werden. Wenn keine Bedingung angegeben wird, findet die Erweiterung immer statt. Mit dem erweiternden Anwendungsfall muss nicht zwingend ein Akteur verbunden sein. Ist dies der Fall, kann er mit dem Stereotyp «secondary» bezeichnet werden.

Das Beispiel „Geld abheben“ zeigt den Use Case *Bargeld abheben* in Rechtecknotation⁵. Der Use Case beinhaltet zwei Erweiterungspunkte⁶. Beide Erweiterungspunkte beschreiben, unter welcher Bedingung die Erweiterung ausgeführt wird. Die Erweiterungsbeziehung enthält eine Einschränkung (Constraint: {Papier vorhanden}). Der Erweiterungspunkt muss eintreten und die Einschränkung muss erfüllt sein, erst dann wird der erweiternde Use Case ausgeführt!

Wie bei der «include» Beziehung wird auch bei der «extend» Beziehung im Diagramm kein Zeitpunkt angegeben, wann der erweiternde Use Case ausgeführt wird. Der Zeitpunkt kann ebenfalls im Use Case Szenario bzw. in einem Verhaltensdiagramm definiert werden.

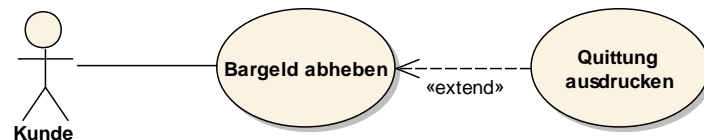


Abb. 11: Beispiel «extend» Beziehung

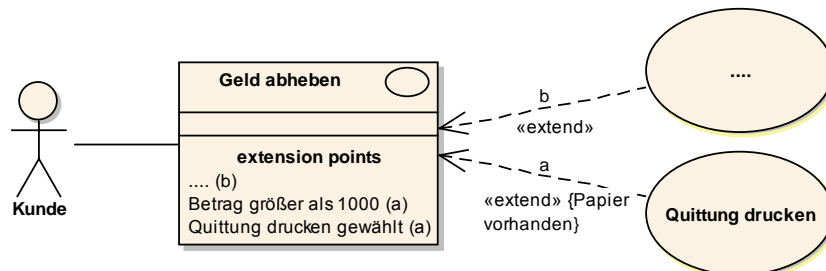


Abb. 12: Beispiel «extend» Beziehung mit Extension Point & Condition

Falls ein Use Case von mehreren Use Cases erweitert wird, kann durch Angabe eines Zusatzbuchstabens eine Beziehung zwischen Erweiterungspunkt und «extend» Beziehung erstellt werden (siehe Abb. 12, Zusatz (a) und (b)).

Hinweis: Bei der Verwendung von Erweiterungsbeziehungen ist darauf zu achten, dass nur Use Cases gleichen Abstraktionsniveaus beschrieben werden. Das Erweitern verleitet dazu, immer tiefer und detaillierter in das zu beschreibende System einzutauchen.

Merke: Das Verhalten von Use Cases kann durch «extend» Beziehungen erweitert werden. Ist ein Erweiterungspunkt definiert (*Extension point*) wird bei dessen Eintreten eine eventuell vorhandene Bedingung (*Constraint*) überprüft und anschließend der erweiternde Use Case ausgeführt.

⁵ Kontextmenü des Elements -> Advanced -> Use Rectangle Notation

⁶ Kontextmenü des Elements -> Advanced -> Edit Extension Points...

Spezialisierung (Generalisierung)

Ein Anwendungsfall (oder auch ein Akteur) kann durch weitere Anwendungsfälle (oder Akteure) spezialisiert werden. Beispielsweise ist der Verkauf an der Abendkasse mit dem Verkauf im Internet bis auf den Vertriebsweg ähnlich. Es bietet sich an, einen generellen Anwendungsfall „Verkaufen“ zu erstellen und in der Spezialisierung dieses Anwendungsfalls die geänderten Abfertigungsschritte, die durch die verschiedenen Vertriebswege entstehen, unterzubringen.

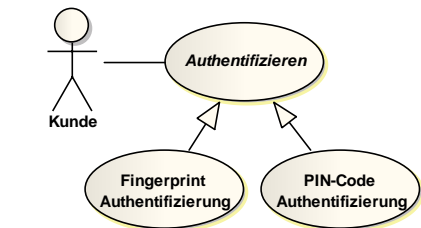


Abb. 13 Generalisierung von Use Cases

Generalisierungsbeziehungen werden auch eingesetzt, um Funktionalitäten allgemein und abstrakt zu beschreiben. Der Use Case *Authentifizierung* in obiger Abbildung Abb. 14 ist abstrakt⁷ und kann selbst nicht ausgeführt werden. Die beiden Verfeinerungen *Fingerprint Authentifizierung* und *PIN-Code Authentifizierung* sind zwei konkrete Varianten des allgemeinen Use Cases. *Authentifizierung* kann als „Platzhalter“ verwendet werden, um zu verdeutlichen, dass sich Kunden authentifizieren müssen und eine der beiden Varianten gewählt werden kann. Der abstrakte Use Case *Authentifizierung* enthält eine allgemeine Beschreibung darüber, wie eine Authentifizierung durchgeführt wird. Die konkreten Use Cases beschreiben die Abweichung des generelleren Falls, wie im oberen Beispiel des Use Cases „Verkaufen“ beschrieben ist.

Ein Akteur beschreibt eine Rolle, diese kann beliebig abstrakt definiert sein! Ein Kunde einer Bank kann z. B. den Use Case *Geld abheben* durchführen. Falls die Bank, von der Geld abgehoben wird, die Hausbank des Kunden ist, soll er auch *Geld einzahlen* dürfen.

Dies kann durch einen weiteren Akteur (*Kunde der eigenen Bank*) beschrieben werden. Da der *Kunde der eigenen Bank* auch ein *Kunde* ist, darf er natürlich alles, was ein *Kunde* darf, somit auch *Geld abheben*.

Dies kann durch eine Generalisierung zwischen den Akteuren *Kunde* und *Kunde der eigenen Bank* geschehen (*Kunde der eigenen Bank* zeigt zum generelleren Akteur *Kunde*). Der *Kunde der eigenen Bank* ist somit auch ein *Kunde* (Generalisierung wird auch als is-a Beziehung bezeichnet), daher erbt der *Kunde der eigenen Bank* auch die Beziehung zum Use Case *Geld abheben* vom *Kunden*. Der Akteur *Kunde* hingegen darf den Use Case *Geld einzahlen* nicht ausführen!

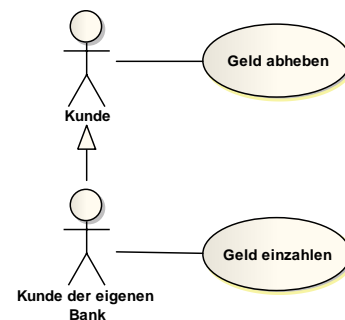


Abb. 14 Generalisierung von Akteuren

Merke: Die Generalisierung zeigt immer in Richtung des generelleren Elements, daher der Name „Generalisierung“. Bei dieser Art der Verbindung spricht man auch von einer is-a Beziehung, da alles vom generelleren Element „geerbt“ wird. Der *Kunde der eigenen Bank* ist also auch ein *Kunde*.

Beschreibungen und Notizen

UML gestattet für alle Anwendungsfälle und Akteure, detaillierte Beschreibungen in Form von verbalen Formulierungen anzufügen. Alternativ können Verhaltensmodelle verwendet werden, um Details in strukturierter Form anzufügen. Notizen können den Diagrammen hinzugefügt werden, die auf wesentliche Gestaltungsüberlegungen hinweisen. Notizen werden mit einem Rechteck dargestellt, deren rechte obere Ecke eingeknickt ist. Eine gestrichelte Linie stellt die Verbindung zwischen der Notiz und dem zu erklärenden Element her. Um Doppelgleisigkeiten zwischen den in den Diagrammen aufscheinenden Notizen und Angaben in den Elementen zu vermeiden, wurde auch vorgesehen, interne Inhalte zitieren zu dürfen⁸.

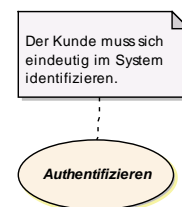


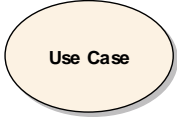
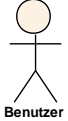

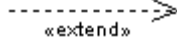
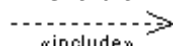

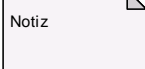

Abb. 15 Notizen

⁷ Use Case selektieren, im Fenster [View / Element Properties] im Abschnitt *Advanced Abstract* auf True setzen

⁸ Element selektieren | Add | Note | OK (leer lassen); Rechtsklick auf den Konnektor, *Link this Note to an Element Feature...*

Grafische Elemente

Die folgende Tabelle listet die Symbole zur Modellierung eines Anwendungsfalldiagramms auf.

Name/Symbol	Verwendung
Anwendungsfall 	Ein Anwendungsfall wird mit einer Ellipse dargestellt, die den Namen des Anwendungsfalls enthält. Der Name des Use Case wird gewöhnlich durch ein Hauptwort und ein Zeitwort gebildet, wodurch das manipulierte Objekt und die durchgeführte Tätigkeit kurz und präzise beschrieben werden. Wenn alternativ die Rechteckschreibweise verwendet wird, können weitere Inhalte eingeblendet werden.
Akteur 	Ein Anwendungsfall wird durch einen Akteur ausgelöst. Die Darstellung entspricht einem Strichmännchen. Man kann einen Akteur auch in einem Rechteck darstellen und das Stereotyp «Actor» über dem Namen des Akteurs angeben.
Verwendet 	Ein Akteur steht in einer Beziehung zum Anwendungsfall, wenn dieser ihn auslöst. Diese Beziehung wird mit einer Verbindungslinie zwischen dem Anwendungsfall und dem Akteur dargestellt.
erweitert 	Wird ein Anwendungsfall durch einen Zweiten unter einer bestimmten Bedingung erweitert, wird diese Beziehung durch die Verbindung der Anwendungsfälle mit einem Pfeil gekennzeichnet, der mit dem Stereotyp «extend» beschriftet wird. Die Pfeilspitze zeigt auf den Anwendungsfall, der erweitert wird.
enthält 	Ist ein Anwendungsfall in einem Zweiten enthalten, d. h. ist er fester Bestandteil von diesem, werden beide Anwendungsfälle mit einem Pfeil verbunden, der das Stereotyp «include» als Beschriftung erhält. Die Pfeilspitze zeigt auf den enthaltenen Anwendungsfall.
Generalisierung 	Diese Beziehung kann zwischen Akteuren und zwischen Anwendungsfällen modelliert werden und bedeutet, dass ein Anwendungsfall oder ein Akteur spezialisiert wird. Die Pfeilspitze zeigt auf den Akteur oder Anwendungsfall, der spezialisiert wird.
Notiz  Notizverbindung 	Notizen sind Diagrammelemente, die an anderen Modellierungselementen angebracht werden. Sie enthalten Informationen zum Verständnis des Modells und werden durch eine unterbrochene Verbindungslinie mit dem Element verbunden.

Beispiel

Ein Kunde möchte mit der Bankomatkarte Geld am Automaten abheben. Der Akteur *Kunde* charakterisiert die Rolle des Kunden und ist die Generalisierung für die Akteur-Rolle *Kunde der eigenen Bank*. Der spezialisierte Akteur *Kunde der eigenen Bank* kann über die Rolle *Kunde* den Anwendungsfall *Authentifizieren* ausführen, der für beide Kundenarten gleichermaßen abläuft. Dieser Anwendungsfall enthält den Anwendungsfall *Konto- und Pin-Kontrolle*, bei dem die Berechtigung des Kunden zur Kartennutzung überprüft wird. Wurde mehrfach eine falsche PIN eingegeben (Constraint: {3x falsch angemeldet}), wird die Karte eingezogen. Um dies zu modellieren, wird der Anwendungsfall *Authentifizieren* mit dem Anwendungsfall *Karte einziehen* erweitert. Dieser wird nur unter der Bedingung, dass der Kunde sich mehrfach nicht identifizieren konnte, abgearbeitet.

Der Akteur *Kunde der eigenen Bank* kommuniziert direkt (nicht über die Rolle *Kunde*) mit dem Anwendungsfall *Geld einzahlen*. Der Kunde hingegen hat keine Beziehung zu dem Use Case *Geld einzahlen* und darf dies somit auch nicht tun.

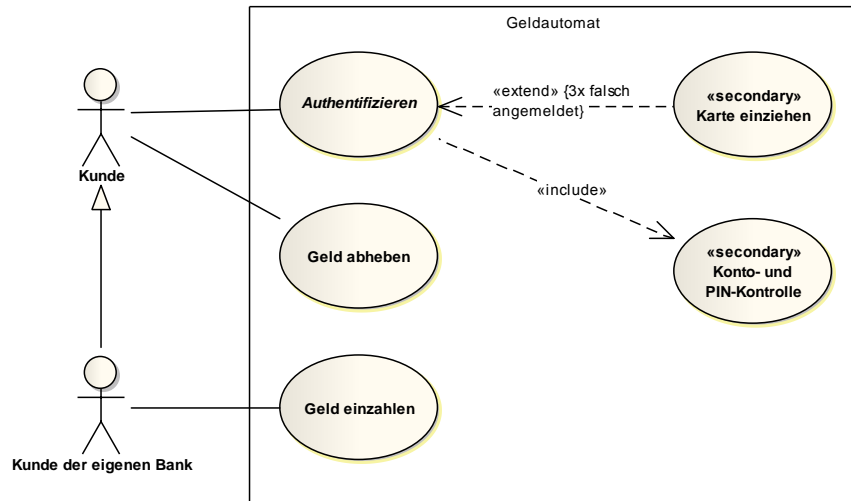


Abb. 16 Beispiel Use Case Diagramm

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Das Use Case Diagramm

- [a] stellt Anwendungsfälle einer Software im zeitlichen Kontext dar.
- [b] zeigt die Struktur von Elementen mit Methoden und Attributen.
- [c] wird verwendet, um Anforderungen an eine Software zu definieren und zu kommunizieren.

2. Eine Person, die eine Software bedient,

- [a] schlüpft in eine Rolle, die durch einen Akteur repräsentiert wird.
- [b] wird im Use Case Diagramm namentlich genannt.
- [c] steht in den Interaktionsdiagrammen im Mittelpunkt.

3. Ein Anwendungsfall wird repräsentiert durch

- [a] ein Quadersymbol mit Namen oberhalb.
- [b] eine Ellipse mit Namen innerhalb oder unterhalb.
- [c] ein Rechteck mit einem Doppelpunkt vor dem Namen.

4. Eine Include-Beziehung sagt, dass

- [a] ein Use Case unbedingt in einem anderen vorkommen muss.
- [b] ein Use Case möglicherweise in einem anderen vorkommen kann.
- [c] ein Use Case eine Spezialisierung eines anderen darstellt.

5. Eine Extend-Beziehung sagt, dass

- [a] ein Use Case unter Umständen mit einem anderen Use Case vorkommen kann
- [b] ein Use Case immer zusammen mit einem anderen Use Case ausgeführt werden muss
- [c] ein Use Case vom anderen abhängt und daher nicht erweitert werden darf

Richtige Antworten: 1c, 2a, 3b, 4a, 5a

Aktivitätsdiagramm (Activity Diagram)

Mit Aktivitätsdiagrammen können zeitliche Abläufe beschrieben werden. Damit ist es möglich Prozesse, Workflows und Algorithmen auf verschiedenen Abstraktionsniveaus zu beschreiben. Häufig werden Aktivitätsdiagramme zur näheren Beschreibung von Use Cases (Anwendungsfälle) eingesetzt. Use Cases können auch mit natürlicher Sprache, sogenannten Szenarien, beschrieben werden, allerdings bleibt dabei die Übersicht nur bei sehr einfachen Abläufen erhalten. Mit Aktivitätsdiagrammen hingegen ist es möglich, auch sehr komplexe Abläufe mit vielen Ausnahmen, Varianten, Sprüngen und Wiederholungen noch übersichtlich und verständlich darzustellen. In der Praxis ist es heute üblich, textuelle Szenarien als Diagramme aufzulösen, um die darin enthaltenen Aussagen als ansprechbare Elemente im Modell verfügbar zu haben. Neben Aktivitätsdiagrammen können auch andere Verhaltensdiagramme, wie das Zustandsdiagramm, Sequenzdiagramm, etc., zur Beschreibung von Use Cases verwendet werden.

Hinweis: Die Semantik der einzelnen Modellelemente unterscheidet sich teilweise trotz gleicher Bezeichnungen erheblich von den Modellelementen in UML 1.x. Das Aktivitätselement von UML 1.x ist der Aktion gewichen, während ein ganzes Aktivitätsmodell nun Aktivität genannt wird. Einige Hinweise zu den UML-Versionen finden sich am Ende des Kapitels über Aktivitätsdiagramme.

Aktivität

Die Aktivität (*Activity*) beschreibt die Ablaufreihenfolge von Aktionen. Sie wird durch ein Rechteck mit abgerundeten Ecken dargestellt. In dem Rechteck befinden sich die Knoten und Kanten der Aktivität. Die Knoten der Aktivität sind in der Regel Aktionen. Es gibt eine Menge verschiedener Aktionen. Am häufigsten wird die „normale“ Aktion verwendet. Zwei weitere wichtige Aktionen sind die *CallBehaviourAction* und *CallOperationAction*, um innerhalb einer Aktivität ein Verhalten aufzurufen, welches anderswo definiert ist. Es ist aber auch erlaubt, dass innerhalb einer Aktivität weitere Aktivitäten gezeichnet werden – Substrukturierung.

Wie jedes Verhalten (Behavior) in der UML kann auch eine Aktivität Parameter haben. Ein- oder ausgehende Objekte einer Aktivität werden als Parameter der Aktivität bezeichnet. Diese Objekte werden auf dem Rechteck der Aktivität platziert und optional unterhalb des Namens der Aktivität mit Typangabe aufgelistet.

Das folgende Beispiel zeigt eine Aktivität zur Produktion von Sechserpacks. Die Aktivität hat zwei Parameter: einen Eingangsparameter *Produzierte Flaschen* im Zustand [leer] und einen Ausgangsparameter *Neues Sechserpack*. Die genaue Deklaration der Aktivitätsparameter steht links oben direkt unter dem Namen der Aktivität.

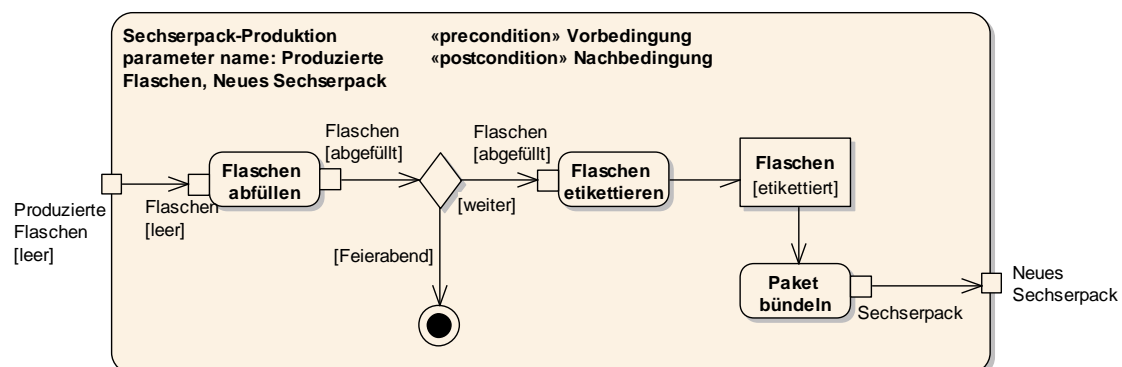


Abb. 17: Beispiel für eine Aktivität „Produktion von Sechserpacks“

In der Aktivität befinden sich verschiedene Arten von Knoten und Kanten.

Die Rechtecke mit den abgerundeten Ecken innerhalb der Aktivität sind Aktionen. Die kleinen Rechtecke an den Aktionen sind sogenannte Pins. Sie stellen die eingehenden bzw. ausgehenden Objekte für die Aktionen bereit.

Tokenkonzept für Aktivitätsdiagramme

Bis UML 1.x waren Aktivitätsdiagramme als Mischung von Zustandsdiagrammen, Petrinetzen und Ereignisdiagrammen definiert, was zu allerlei theoretischen und praktischen Problemen führte.

Seit der UML 2.x liegt den Aktivitätsdiagrammen eine aus den Petrinetzen entlehnte *Tokensemantik* zugrunde, mit der präzise Regeln für den Ablauf- und Objektfluss, inklusive Parallelisierung und Zusammenführung geschaffen wurden. Ein Token entspricht dabei genau einem Ablauf-Thread, der erzeugt und vernichtet werden kann. Dabei repräsentiert das Token entweder das Fortschreiten des Ablauf- oder des Datenflusses. Durch die formale Spezifikation der Semantik der Aktivitätsdiagramme besteht nun die Möglichkeit, eine automatische Verifikation durch Simulation von Aktivitätsdiagrammen durchzuführen.

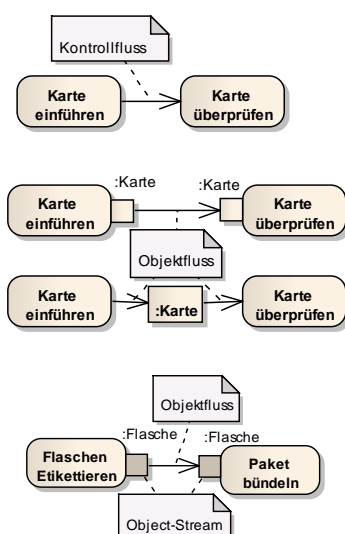
Durch die Überarbeitung des UML Aktivitätsdiagramm haben sich folgende Begriffe geändert:

- Die einzelnen elementaren, nicht teilbaren Schritte in einem Ablauf heißen nicht mehr *Aktivitäten*, sondern *Aktionen*.
- Eine Menge von Schritten, also letztendlich ein Ablaufdiagramm bzw. Teilablauf, wird nun *Aktivität* genannt.
- Während bis UML 1.x jede eingehende Transition einen Ablaufschritt gestartet hat, ist jetzt eine implizite Synchronisation vorhanden, d. h., alle eingehenden Objekt- und Kontrollflüsse müssen vorliegen, damit die Aktion startet.
- Ebenso wird ein Aktionsknoten erst dann verlassen, wenn alle ausgehenden Kanten feuern können. In UML 1.x war es genau umgekehrt, wenn mehrere ausgehende Kanten (früher Transitionen genannt) notiert waren, musste über entsprechende Bedingungen sichergestellt werden, dass stets nur eine Kante feuern kann. Jetzt wird gewartet, bis alle Bedingungen für alle ausgehenden Kanten erfüllt sind, bevor gefeuert wird.
- Es existieren einige neue Elemente:
 - Aktivitäten können Objektknoten als Ein- und Ausgangsparameter haben.
 - Es können Vor- und Nachbedingungen für Aktivitäten definiert werden.
 - Anfangs- und Endaktivität heißen jetzt Startknoten und Endknoten.

Verbindungen

Die Verbindungen zwischen Aktionen werden in Kontrollfluss- und Objektfluss-Kanten unterschieden (*ControlFlow* bzw. *ObjectFlow*). In der Notation sind beide Kanten gleich: eine durchgezogene Linie mit offener Pfeilspitze.

Unterschieden werden Objektfluss-Kanten explizit, indem sie zwischen *Object-Pins* oder *Activity-Nodes* gezogen werden (kleines Rechteck an Aktion oder Aktivität), zu einem oder von einem *Datastore*, *Central Buffer Node* oder Objekt führen.



Ein Kontrollfluss verbindet Aktionen und Aktivitäten. Nachdem *Karte einlesen* abgeschlossen ist, fließt das gedachte Token entlang des Kontrollflusses, falls *Karte überprüfen* bereit ist, aktiviert zu werden!

Bei einem Objektfluss werden zusätzlich zur Kontrolle auch Daten übermittelt. Falls mehrere Objekttoken ankommen, werden diese standardmäßig nach FIFO weiter gereicht. Alternativ zur Pin-Notation kann auch ein Objekt verwendet werden.

Ein Spezialfall von Objektflüssen sind *Object-Streams*. Dabei handelt es sich um einen kontinuierlichen Fluss an Daten (Objekten). Vergleichbar mit einem Förderband, auf dem stetig Flaschen abgefüllt werden.

Objektflüsse und Kontrollflüsse können auch aufgeteilt werden. Ein *Central Buffer Node* oder ein *Datastore* können verwendet werden, um Daten vorübergehend bzw. permanent zu speichern.

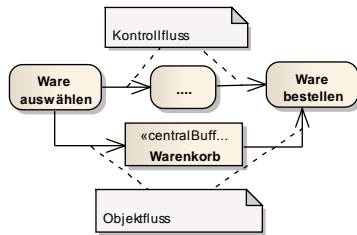


Abb. 18 Kontrollfluss / Objektfluss

Waren können im Warenkorb (*Central Buffer Node*) zwischengespeichert werden und später wieder abgerufen werden. Falls der Prozess vorher beendet wird, werden zwischengespeicherte Daten vernichtet – im Gegensatz zum *Data Store*.

Verzweigungen

Eine Verzweigung des Prozesses wird mittels Rautensymbol (*Decision*) erreicht. Eine *Decision* kann beliebig viele ausgehende Kanten haben (in der Regel mindestens 2). Falls mehrere ausgehende Kanten vorhanden sind, wird dies als *Switch* interpretiert. Alternativ kann ein *Switch* durch mehrere hintereinander geführte *Decisions* ausgedrückt werden, dies ist allerdings ein schlechter Modellierungsstil! Zu beachten ist, dass jede ausgehende Kante einer *Decision* eine Wächterbedingung (*Guard*) besitzt.

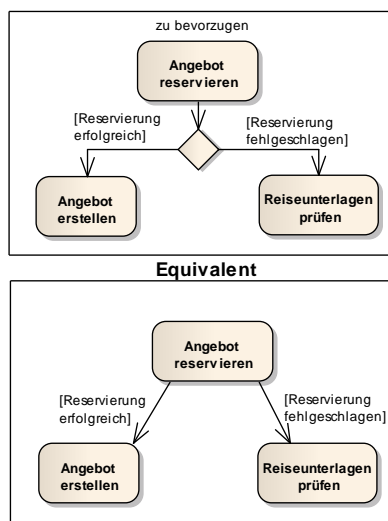


Abb. 19 Explizite vs. Implizite Decision

Hinweis: Wächterbedingungen müssen vollständig sein ($x < 0$ und $x > 0$; Fehler bei $x == 0$) und dürfen sich nicht überlappen ($x \leq 0$ und $x \geq 0$; Fehler bei $x == 0$).

Alternativ zur *Decision* können die ausgehenden Kanten mit ihren Wächterbedingungen direkt aus einer Aktion/Action führen.

Achtung: Mehrere, nicht beschriftete Ausgänge aus einer Aktion/Activity bedeuten jedoch *Splitting* (Parallelisierung). Da dies leicht verwechselt werden kann, wird üblicherweise das Rautensymbol für Verzweigung (*Decision*) gesetzt, wodurch klargestellt ist, dass nur ein einziger Ausgang gewählt wird (disjunkt).

Zusammenführen

Durch eine *Decision* wird im Prozess ein alternativer Weg ausgewählt. Besteht der Bedarf Schleifen zu definieren, ist es notwendig, ein *Merge* Element zu verwenden!

Wird auf ein *Merge* Element verzichtet und die rückführende Kante direkt in eine Action/Activity geführt, impliziert dies die Zusammenführung zweier nebenläufiger Prozesse (impliziter *Join*). Wird also die rückführende Kante direkt in Termin auswählen geführt, muss an jeder Kante ein Token anliegen. Dies wird in diesem Beispiel nie passieren und führt zu einem blockierten Prozess (*dead-Lock*). Zwei eingehende Kanten in eine Action/Activity sind nach UML nicht verboten (siehe Abb. 21), die implizite *Join* Semantik muss allerdings bedacht werden. Um falsche Interpretationen zu vermeiden, sollte auf implizite UML Semantik gänzlich verzichtet werden!

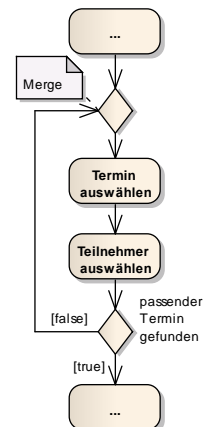


Abb. 20 Mergen

Splitting (Parallelisierung) und Synchronisation

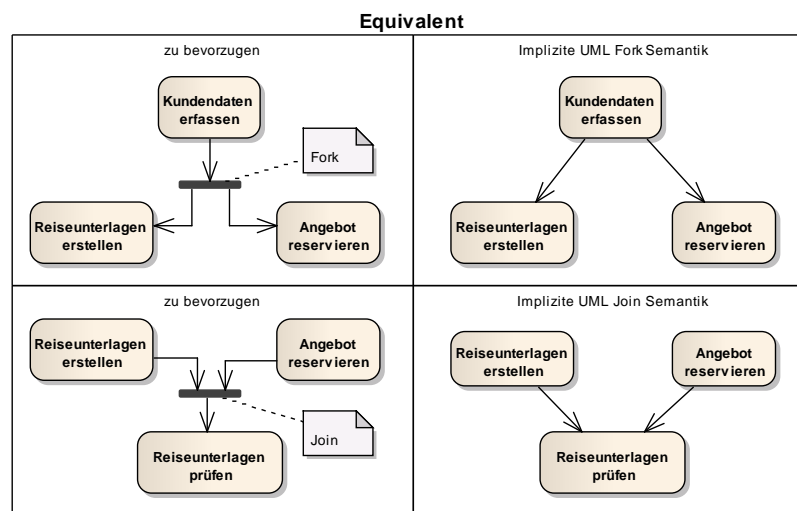


Abb. 21 Splitting und Synchronisation

Mit einem Splitting Knoten (*Fork Node*) wird ein Prozessfluss parallelisiert, um nebenläufige Prozesse zu modellieren. Nebenläufigkeit heißt zeitlich unabhängig, muss allerdings nicht zwangsläufig gleichzeitig bedeuten. *Reiseunterlagen erstellen* und *Angebot reservieren* (Abb. 21) können, müssen aber nicht gleichzeitig durchgeführt werden. Ein Fork darf auch mehr als zwei abgehende Kanten haben.

Beim *Fork Node* wird jedes eingehende Token dupliziert, auf jeder abgehenden Kante beginnt ein Token zu laufen. Kann ein Token an einer ausgehenden Kante nicht weiter gereicht werden, wird es in einer FIFO-Liste zwischengespeichert, bis die nachfolgende Aktion/Aktivität dieses Token aufnehmen kann. Dies ist eine Ausnahme bei UML Aktivitätsdiagrammen, da in der Regel an „pseudo“-Knoten keine Tokens stehen bleiben.

Nebenläufige Prozesse können mittels Synchronisation (*Join Node*) zusammengeführt werden. Dazu muss an jeder eingehenden Kante ein Token (Kontroll- oder Objekttoken) anliegen. Nach dem Zusammenführen werden alle anliegenden Kontrolltoken und identische Objekttoken zu einem verschmolzen. Im Gegensatz dazu werden alle anliegenden Objekttoken weitergereicht! Lediglich identische Objekttoken werden verschmolzen und nur einmal weiter gereicht.

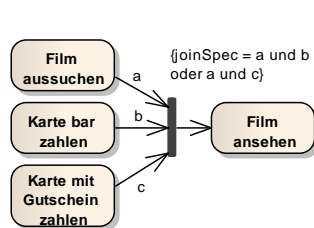


Abb. 22 Join Specification

Der *Join Node* hat eine Implizite UND Semantik. Wenn lediglich eine Auswahl an nebenläufigen Prozessen vollendet sein muss, um den Prozess synchron fortzuführen, bietet die UML die *Join Specification (JoinSpec)*. Damit besteht die Möglichkeit eine Bedingung zu definieren, welche das Zusammenführen beschreibt. Im Beispiel muss an der Kante *a und b* oder *a und c* ein Token anliegen. Der Film muss auf jeden Fall ausgesucht werden, wie bezahlt wird, ist egal.

Schachteln von Aktivitätsdiagrammen

Aktivitäten bestehen in der Regel aus Aktionen, welche die einzelnen Schritte einer Aktivität beschreiben. Dies ist vergleichbar mit einer Operation/Funktion einer Programmiersprache und den einzelnen Anweisungen in der Operation. Genauso wie eine Operation andere Operationen aufrufen kann, besteht diese Möglichkeit auch bei Aktionen in Aktivitäten.

Das Schachteln hilft a) mit dem üblichen A4-Format (auch über mehrere Seiten hinweg) auszukommen und b) die Inhalte so zu gliedern, dass sie eine für den jeweilig verantwortlichen Freigeber passende Detailtiefe aufweisen.

Das Aufrufen einer Aktivität wird durch eine *Call Behavior Action* durchgeführt.

Call Behavior Actions sind grafisch durch ein Gabel-Symbol in der rechten unteren Ecke von anderen Aktionen unterscheidbar.

In Abb. 23 wird in der Aktivität *Geld abheben* mehrmals die Aktivität *Karte ausgeben* aufgerufen. Karte ausgeben kann wiederum durch beliebige *Actions* beschrieben worden sein.

Hinweis: Durch *Call Behavior Actions* kann das Problem von duplizierten Aktionen/Aktivitäten vermieden werden. Der Trick ist, eine Aktivität zu definieren, welche durch *Call Behavior Actions* beliebig oft aufgerufen werden kann. Zu beachten ist, dass lediglich Aktivitäten (*Activities*) durch *Call Behavior Actions* aufgerufen werden können.

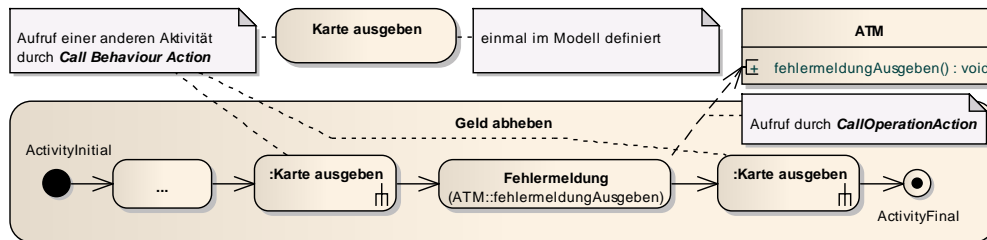


Abb. 23 Aufruf einer Aktivität mittels Aktion

Call Operation Actions sind vergleichbar mit *Call Behavior Actions*, sie rufen allerdings kein Verhalten (Aktivität) auf, sondern direkt Operationen, die anderswo definiert sind (z. B. Operationen einer Klasse). Durch die Aktion *Fehlermeldung* in Abb. 23 wird z. B. die Operation *fehlermeldungAusgeben()* der Klasse *ATM* aufgerufen.

Durch *Call Operation Actions* können Aktivitätsdiagramme und Strukturdiagramme, wie das Klassendiagramm, explizit verbunden werden und somit definiert werden, wo ein bestimmtes Verhalten realisiert wird. Einerseits wird das von Qualitätssystemen (SPICE, CMMI, ...) eingefordert, andererseits senkt der geringe Aufwand, den Verweis anzulegen, den Aufwand bei späteren Änderungen dramatisch!

In Enterprise Architect besteht die Möglichkeit, Elemente zu strukturieren. Ein *strukturiertes (composite) Element* besitzt einen Link zu einem Diagramm in dem weiterführende Informationen definiert sind. Grafisch werden verlinkte Elemente mittels Brille/Kette in der rechten unteren Ecke dargestellt.

Diese Möglichkeit ist nicht in der UML-Spezifikation definiert, bietet aber eine gute Strukturierungsmöglichkeit von Diagrammen.

Mit dieser Kaskadierung von Diagrammen behält man auch bei komplexen Abläufen die Übersicht. Diese Untergliederung in Sub-/Detailmodelle kann hilfreich und auch notwendig sein,

- um hinreichende Unterteilung um Standardpapierformat einhalten zu können und
- zur Erzeugung von Detailgliederungen, die in verschiedene Dokumente eingebunden werden und von verschiedenen Verantwortungsträgern freigegeben werden.

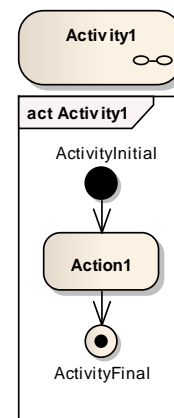


Abb. 24 Strukturierte Aktivitäten

Verantwortlichkeitsbereiche (Swimlanes)

Einzelne Aktionen im Aktivitätsdiagramm werden im Normalfall von einem zuständigen *Actor* durchgeführt. Um in einem Diagramm, die Zuordnung der einzelnen Aktionen zu den Akteuren darstellen zu können, gibt es die Möglichkeit, sogenannte *Swimlanes* (Verantwortlichkeitsbereiche) einzuführen. Diese senkrecht oder waagrecht verlaufenden Bahnen symbolisieren den *Actor* und stellen die Aktivitäten durch die grafische Zuordnung der einzelnen Aktionen zu einer Bahn in dessen Verantwortlichkeitsbereich. Alternativ zur *Swimlane* können auch *Partitions* verwendet werden. *Partitions* sehen *Swimlanes* sehr ähnlich, sind aber Teil des Modells und nicht nur im Diagramm eingezeichnet. Die nachfolgende Abb. 25 zeigt die Verwendung von *Partitions*.

Asynchrone Prozesse

Durch Kontrollflüsse und Objektflüsse werden Aktivitäten und Aktionen verbunden. Die so definierten Prozesse sind „synchron“, d. h., es ist determiniert, welche nachfolgenden Schritte möglich sind, alternativen werden ebenfalls direkt mittels *Decision* modelliert.

Durch Verwendung von Signalen (*Send Signal Action*, *Receive Signal Action* und *Timer Action*) ist es möglich, zwei oder mehrere Prozesse zu entkoppeln. Durch ein *Send Signal* wird ein Broadcast Signal gesendet. Alle *Receive Signals*, für welche dieses Signal bestimmt ist, werden aktiv. Für eine bessere Lesbarkeit können Abhängigkeiten durch *Dependency* Kanten vom *Receive Signal* zum *Send Signal* Element definiert werden.

Das nebenstehende Beispiel zeigt, dass die Aktion *Pizza bestellen* (eine *Send Signal Action*) ausgeführt wird und das Signal zum Backen der Pizza in der Pizzeria von der Aktion (eine *Receive Signal Action*) *Bestellung entgegennehmen* aufgefangen wird.

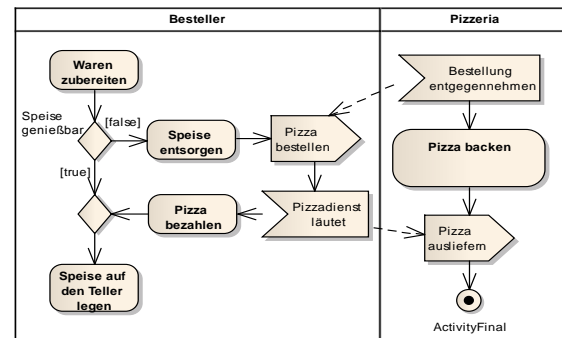
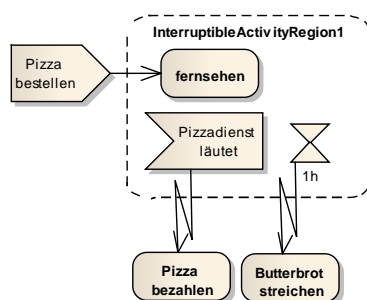


Abb. 25 Send/Receive

Hinweis: *Receive Signal Actions* brauchen nicht zwangsläufig eine eingehende Kante. Die Regel besagt, dass alle Aktivitäten/Aktionen welche keine eingehende Kante besitzen mit einem Token belegt werden. Bei Unterbrechungsbereichen verhalten sich *Signal Actions* unterschiedlich und werden nur aktiv, sobald der Unterbrechungsbereich betreten wird!

Unterbrechungsbereich



Der Unterbrechungsbereich (*Interruptible Activity Region*) definiert einen speziellen Bereich eines Prozesses. Wird der Unterbrechungsbereich betreten, kann der ausgeführte Prozess zu jedem beliebigen Zeitpunkt unterbrochen werden und ein alternativer Pfad gewählt werden.

Der Unterbrechungsbereich kann durch Eintreffen eines Signals (*Receive Signal*) oder durch Ablauf eines Zeit-Events (*Time Event*) unterbrochen werden. Eine Unterbrechung des Prozesses wird durch einen Unterbrechungspfeil (*Interrupt Flow*) definiert.


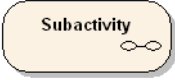
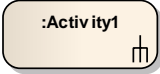
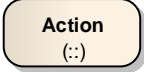
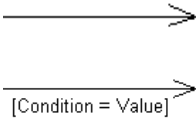

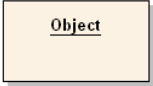
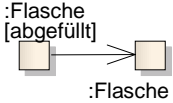
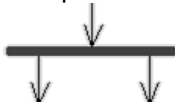
Abb. 26 Unterbrechungsbereich

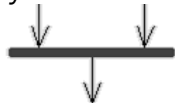



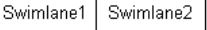

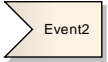

/Objektfluss-Kante in den Unter-


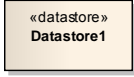
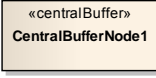
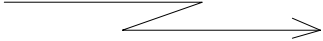
nein, wird der Bereich aktiv und somit auch die *Time Event Actions* und *Receive Signal Actions*. Führt eine Kontroll-/Objektfluss-Kante wieder aus dem Unterbrechungsbereich heraus, wird der Unterbrechungsbereich ordnungsgemäß verlassen und dadurch wieder inaktiv. Solange der Unterbrechungsbereich aktiv ist, kann er über Unterbrechungspfeile an beliebigen Stellen verlassen werden. Eventuell ausgeführte Aktivitäten/Aktionen werden gestoppt. Die UML-Spezifikation enthält keine Beschreibung darüber, wie bereits angefangene Aktivitäten/Aktionen behandelt werden, falls sie unterbrochen werden. Eine Best-Practice Interpretation ist, dass ein Rückgängigmachen (Rollback) der teilweise durchgeführten Aktionen/Aktivitäten durchgeführt wird. Hinweis: Der Interrupt-Flow muss von einem Element in der Interruptregion auf ein Element außerhalb der Region zeigen.

Im obigen Beispiel in wird der Unterbrechungsbereich nach der Bestellung einer Pizza betreten. Während auf die Pizza gewartet wird, wird die Aktivität *fernsehen* ausgeführt. Da *fernsehen* keine ausgehende Kante besitzt, würde der Prozess hier stehen bleiben! Der Unterbrechungsbereich, und somit auch die Aktivität *fernsehen*, wird unterbrochen, sobald die Pizza geliefert wird, oder länger als eine Stunde gewartet wurde.

Grafische Elemente

Name/Symbol	Verwendung
Aktion 	Das Symbol der Aktion besteht aus einem Rechteck mit abgerundeten Ecken. Per Definition ist eine Aktion ein einzelner Schritt, der nicht mehr unterteilt werden kann und auch nicht durch äußere Einflüsse unterbrechbar ist.
Strukturierte Aktivität 	Das Symbol der strukturierten Aktivität wird mit dem Symbol der Aktivität dargestellt. Im rechten unteren Bereich ist ein Brillen/Ketten-Symbol angebracht. Dieses Element verlinkt zu einem anderen Diagramm. Nicht Teil des UML Standards, allerdings ein wertvolles Strukturierungskonstrukt.
Call Behavior Action 	Eine <i>Call Behavior Action</i> erlaubt das Aufrufen von beliebigen Verhalten (Aktivitäten) aus einem bestehenden Prozess. Damit kann die redundante Definition von Aktivitäten/Aktionen verhindert werden.
Call Operation Action 	Eine <i>Call Operation Action</i> ruft ein konkretes Verhalten eines Strukturelementes auf, z. B. die Operation einer Klasse. Elementname und Verhaltensname werden durch „::“ getrennt. ([Elementname]::[Verhaltensname])
Kontrollfluss 	Zwei Aktionen werden mit einem Pfeil verbunden, wenn der Aktivitätsfluss von einer zur nächsten Aktion/Aktivität wechselt. Die Pfeilspitze zeigt in die Richtung des Prozessflusses. Der Pfeil kann eine Bedingung als Beschriftung erhalten, wenn der Prozessfluss nur bei dieser Bedingung stattfindet. Dies ist der Fall, wenn mehrere Transitionen aus einer Aktivität herausgehen oder der Fluss durch eine Raute (<i>Decision</i>) aufgeteilt wird.
Verzweigung, Zusammenführung 	Mit dem Rautensymbol (<i>Decision</i>) kann der Prozessfluss verzweigt oder wieder zusammengeführt (<i>Merge</i>) werden. Geht eine Kante ein und mehrere ab, handelt es sich um die Verzweigung (<i>Decision</i>), gehen mehrere Kanten ein und eine ab, handelt es sich um eine Wegezusammenführung (<i>Merge</i>). Bei einer Wegezusammenführung wird in der Regel keine Beschriftung eingesetzt.
Objekt 	Wenn in einem Prozess Daten erzeugt und verwendet werden, können diese als Objekte (Instanzen von Klassen) repräsentiert werden. Das Objekt kann ohne Typ angegeben werden (wie in der Abbildung links). Bei typisierten Objekten steht der Typ-Name nach dem Objekt-Namen. Die Notation ändert sich auf: <u>Objekt-Name: Typ-Name</u>
Objektfluss 	Der Objektfluss beschreibt die Übergabe der Kontrolle von einer Aktivität/Aktion zur nächsten und überträgt zusätzlich zur Kontrolle, Daten (Objekte). Ausgangspunkt und Endpunkt des Objektflusses ist in der Regel ein Objekt. Dies kann ein <i>ObjektNode</i> (kleines Quadrat an Aktivität/Aktion), ein <i>Objekt</i> (Instanz einer Klasse), ein <i>Central Buffer Node</i> (transienter Pufferknoten) oder ein <i>Datastore</i> (persistenter Pufferknoten) sein. <i>ObjectNodes</i> können wie Instanzen einer Klasse einen Typ besitzen (:Flasche) und zusätzlich ihren Zustand definieren ([abgefüllt]).
Splitter 	Durch Splitting (<i>Fork</i>) kann der Prozessfluss in mehrere nebenläufige Prozessflüsse aufgeteilt werden. Beim <i>Fork</i> wird das eingehende Token (Kontroll- oder Daten-Token) vervielfältigt. Jede ausgehende Kante bekommt ihr eigenes Token.

Name/Symbol	Verwendung
<p>Synchronisation</p> 	<p>Durch die Synchronisation (<i>Join</i>) können nebenläufige Prozessflüsse zusammengeführt werden. Dabei findet eine Synchronisation statt, d. h., die Abarbeitung wird so lange angehalten, bis alle Teilflüsse (Token) am Synchronisationselement angekommen sind. Die UND Semantik des <i>Join</i> kann durch <i>Join Specifications</i> redefiniert werden (siehe Abb. 22).</p>
<p>Startpunkt</p> 	<p>Der Startpunkt ist der Ausgangspunkt des Prozesses. Sind mehrere Startpunkte vorhanden, werden die davon betroffenen Zweige des Prozesses nebenläufig gestartet. Falls kein Startpunkt vorhanden ist, werden alle Knoten, die keine eingehenden Kanten haben, als Startpunkte interpretiert. Für ein besseres Verständnis sollte darauf geachtet werden, einen Startpunkt pro Prozess zu definieren.</p>
<p>Endpunkt</p> 	<p>Nachdem alle Aktionen der Aktivität abgearbeitet wurden, endet der Prozessfluss dieser Aktivität. Dieser Punkt wird mit dem Endpunkt dargestellt. Ein Aktivitätsdiagramm darf eine beliebige Anzahl von Endpunkten (<i>Activity Final</i>) enthalten; bei endlos laufenden Prozessen kann er fehlen. Durch die Verwendung mehrerer Endpunkte kann die Terminierung des Prozesses, an unterschiedlichen Punkten im Prozess, besser dargestellt werden. Achtung: Werden Endpunkte innerhalb verschachtelter Aktivitäten verwendet, beendet der Endpunkt nicht den gesamten Prozess, sondern lediglich <u>alle</u> Tokens, die im Subprozess am Laufen sind.</p>
<p>Abbruch</p> 	<p>Abbruch, <i>Flow Final</i> bedeutet, dass ein Token, der dieses Symbol erreicht, vernichtet wird. Der Prozesszweig wird hier abgebrochen. Falls noch weitere Token vorhanden sind, wird der Gesamtprozess weiter ausgeführt, handelt es sich jedoch um den letzten Token, wird der gesamte Prozess beendet.</p>
<p>Swimlanes/Partition</p> 	<p>Möchte man Zuständigkeitsbereiche im Prozessfluss modellieren, z. B. Aktionen/Aktivitäten, welche verschiedenen Paketen/Komponenten/Aktor angehören, kann man die Verantwortlichkeitsbereiche mit senkrechten oder waagrechten Linien modellieren. Der Name des Bereichs, der zwischen zwei Linien liegt, wird im oberen Bereich mit dem Namen des zuständigen Elements beschriftet.</p>
<p>Send Signal Action</p> 	<p>Ein <i>Send-Signal</i> ist eine <i>Action</i>, die verwendet wird, um während der Ausführung eines Prozesses asynchrone Nachrichten an andere Prozesse zu senden.</p>
<p>Receive Signal Action</p> 	<p>Ein <i>Receive Signal</i> ist eine <i>Action</i>, welche auf ein Signal (<i>Event</i>) wartet. Nach Eintreffen des Signals wird die Aktion ausgeführt und der Flow weiter geführt. <i>Receive Events</i> werden verwendet, um Asynchronität zu modellieren. Wenn das <i>Receive Event</i> keine eingehenden Kanten besitzt und das Element, welches das <i>Receive Event</i> beinhaltet aktiv ist, ist es „feuerbereit“.</p>
<p>Time Event</p> 	<p>Ein <i>Time Event</i> erzeugt periodisch einen Output (<i>Token</i>). Der Output triggert weitere Aktionen und kann auch in Zusammenhang mit <i>Interruptable Activity Regions</i> verwendet werden. Wenn das <i>Time Event</i> keine eingehenden Kanten besitzt und das Element welches das <i>Time Event</i> beinhaltet aktiv ist, ist es „feuerbereit“.</p>

Name/Symbol	Verwendung
Interruptibel Activity Region 	Eine <i>Interruptible Activity Region</i> ist ein Bereich, der durch Ereignisse (<i>Receive Events</i> , <i>Time Events</i>) verlassen werden kann. Alle aktuell ausgeführten Aktionen werden unterbrochen und der alternative Weg wird weiter verfolgt.
Datastore 	Ein <i>Datastore</i> ist ein persistenter Pufferknoten. Er wird eingesetzt, um Daten aus Objektflüssen aufzunehmen. Damit kann ausgedrückt werden, dass in einem Prozess auf vorhandene Daten zugegriffen wird, bzw. neue Daten persistent gespeichert werden.
Central Buffer Node 	Ein <i>Central Buffer Node</i> ist ein transienter Pufferknoten. Er verhält sich wie ein <i>Datastore</i> , mit dem Unterschied, dass gespeicherte Daten nach Beendigung der Aktivität, mittels <i>Activity Final</i> , gelöscht werden. Er hat also die Semantik einer lokalen Variablen einer Operation in OO Programmiersprachen.
Interrupt Flow 	Ein <i>Interrupt Flow</i> wird verwendet, um eine <i>Interruptible Activity Region</i> zu verlassen.

Beispiel

Im folgenden Beispiel wird der Prozess zur Vorbereitung einer Feier beschrieben. Die Beschreibung wurde aus Gründen der Übersichtlichkeit in mehreren Diagrammen modelliert.

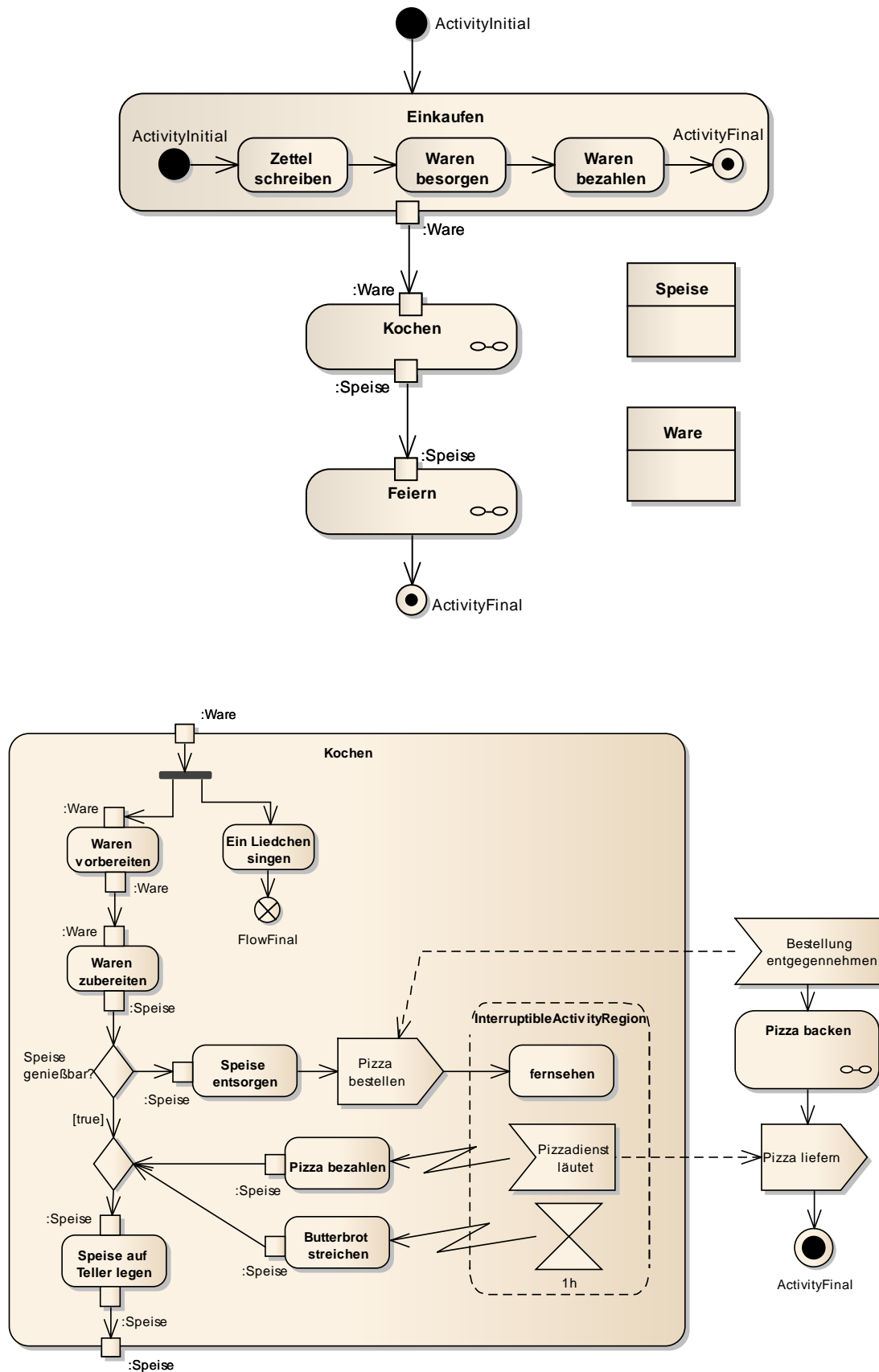


Abb. 27: Beispiel: Prozess zur Durchführung einer Feier

Die obige Abbildung beschreibt drei Aktivitäten, die nacheinander durchgeführt werden (*Einkaufen*, *Kochen*, *Feiern*). Die einzelnen durchzuführenden Aktionen des Einkaufens sind in der Aktivität *Einkaufen* angeführt.

Aus der Aktivität *Einkaufen* fließt ein Objektfluss mit der/den erworbenen *Ware(n)* in die Aktivität *Kochen*.

Die Typen (*Ware*, *Speise*) der Objekte (*Object Nodes*) sind als Klassen modelliert und können somit ebenfalls noch näher beschrieben werden.

Ersichtlich ist, dass *Kochen* *Ware* als Input bekommt und *Speise* weiter reicht.

Kochen ist in einem eigenen Diagramm modelliert. Der Prozess des Kochens beginnt mit nebenläufigen Prozessen. Während gekocht wird, wird auch ein Lied gesungen.

Der nebenläufige Prozess *Ein Liedchen singen* wird mit einem *Flow Final* beendet. Das heißt, wenn die Aktion (Singen des Liedes) beendet ist, „stirbt“ dieser Zweig des Prozesses.

Der zweite nebenläufige Prozess beginnt mit dem Vorbereiten der Waren und dem anschließenden Zubereiten der Waren. In die Aktivität *Waren zubereiten* fließt *Ware* hinein und *Speise* wieder hinaus. Hier werden aus Waren Speisen. Ist die Speise ungenießbar, wird sie entsorgt und eine Pizza wird bestellt.

Das Bestellen der Pizza ist eine *Send Signal Action* und wird von *Bestellung entgegennehmen* (*Receive Signal Action*) aufgefangen. Dieses Signal triggert den Start eines unabhängigen weiteren Prozesses, das Backen der Pizza.

Nachdem die Pizza bestellt wurde, wird die Aktivität *fernsehen* ausgeführt. Dabei wird ein Unterbrechungsbereich (*Interruptable Activity Region*) betreten. Durch das Betreten der Region werden die in dem Unterbrechungsbereich enthaltenen *Receive Signal Action* und *Time Event Action* aktiv, d. h., der Timer beginnt zu laufen und die *Receive Signal Action* ist bereit, Signale aufzufangen.

Hinweis: Befindet sich der Prozessfluss nicht im Unterbrechungsbereich, werden eventuell eintreffende Signale verworfen!

Befinden wir uns im Unterbrechungsbereich und das Signal des *Pizza ausliefern* (*Send Signal Action*) trifft ein, verlassen wir den Unterbrechungsbereich und *fernsehen* wird unterbrochen! Im nächsten Schritt wird die Pizza bezahlt und der Prozessfluss wird zusammengeführt (*Merge Node* nach dem *Decision Node*).

Alternativ wird der Unterbrechungsbereich verlassen, wenn die Pizza nicht innerhalb von einer Stunde eintrifft, dann läuft der Timer ab, der Unterbrechungsbereich wird verlassen und ein Butterbrot wird als Alternative gestrichen. Anschließend wird wieder zusammengeführt.

Nachdem die Speise auf den Teller gelegt wurde, ist die Aktivität *Kochen* beendet.

Durch die *Send Signal Action* *Pizza bestellen* wird asynchron ein weiterer Prozess gestartet. Dieser Prozess wird in einem eigenen Diagramm erstellt und erhöht so die Lesbarkeit des Modells.

Durch Anführen der *Send* und *Receive Signal Action* *Pizza bestellen* und *Pizzadienst läutet*, wird beschrieben, wer auf die gesendeten Signale reagiert.

Die Aktivität *Pizza backen* kann ebenfalls verfeinert werden, indem sie auf ein Diagramm verlinkt, welches den Prozess *Pizza backen* detaillierter beschreibt.

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Das Aktivitätsdiagramm

- [a] zeigt zeitliche Abfolgen von Aktionen und Aktivitäten.
- [b] hat sich seit UML 1.4 nicht wesentlich verändert.
- [c] zeigt Aktionen und Aktivitäten in einer Größe relativ zu ihrer Dauer.

2. Eine Aktivität

- [a] kann hierarchisch verschachtelt weitere Aktivitäten beinhalten.
- [b] ist ein nicht weiter unterteilbarer Schritt in einem Ablauf.
- [c] stellt den Nachrichtenaustausch zwischen Objekten dar.

3. Splitting und Synchronisation werden eingesetzt, um

- [a] Wege aus einer Entscheidung miteinander zu vergleichen.
- [b] die zeitliche Dauer von Aktionen zu betonen.
- [c] Parallelität von Aktionen darzustellen.

4. Swimlanes sind

- [a] Verantwortlichkeitsbereiche, die nach ihrem zuständigen Akteur benannt werden.
- [b] Verantwortlichkeitsbereiche, innerhalb derer Akteure dargestellt werden.
- [c] der bedingte Übergang zwischen zwei Aktionen.

Richtige Antworten: 1a, 2a, 3c, 4a

Zustandsdiagramm (State Machine Diagram)

Zustandsdiagramme sind keine Erfindung der UML, sie gehen auf David Harels Zustandsautomaten, entwickelt in den 80er Jahren, zurück. Diese Darstellungsform wurde in die UML aufgenommen.

Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann, und die Ursachen der Zustandsänderungen. Man kann für ein Objekt den Zustand und die Änderung des Zustandes in Abhängigkeit von ausgeführten Operationen modellieren. Dabei wird besonderer Wert auf die Übergänge von einem Zustand in den nächsten gelegt. Man kann so ein Objekt von der Initialisierung bis zur Freigabe modellieren. Das Zustandsdiagramm beschreibt, durch welche Operationen oder Ereignisse die Zustände des Objekts geändert werden. Weiters ist aus ihnen ersichtlich, welche Belegung die Attribute des Objekts vor dem Übergang besitzen oder besitzen müssen.

Ein Objekt kann als Zustandsdiagramm/-„System“ modelliert werden unter der Voraussetzung, dass Sie eine Liste von Zuständen angeben können, für die gilt:

- Das Objekt befindet sich immer (zu jedem beliebigen Zeitpunkt seiner Existenz) in einem (1) Zustand dieser Liste, anders ausgedrückt:
- Das Objekt befindet sich nie in keinem der genannten Zustände (wenn doch, dann fehlt Ihnen zumindest ein Zustand in der Liste).
- Nie in mehreren Zuständen Ihrer Liste zugleich (wenn doch, dann haben Sie die Untergliederung in Zustände falsch gewählt).

Ein Objekt in einem Zustand kann dort ruhen, es ist aber auch möglich, in Zuständen „Aktivität“ vorzusehen.

Befindet sich ein Objekt in einem Zustand, dann können durchaus auch Subzustände für diesen Zustand modelliert werden, z. B. in einem untergeordneten Diagramm (Composite Element/Child Diagramm). Ist das Verhalten in einem Zustand prozeduraler Natur, dann kann das Subdiagramm natürlich auch ein Verhaltensdiagramm anderer Art sein.

Zustandsdiagramme müssen einen Anfangszustand und können einen Endzustand haben. Zustandsübergänge, sogenannte Transitionen, werden stets durch ein Ereignis ausgelöst (z. B. Bedingung, Time-out, ...).

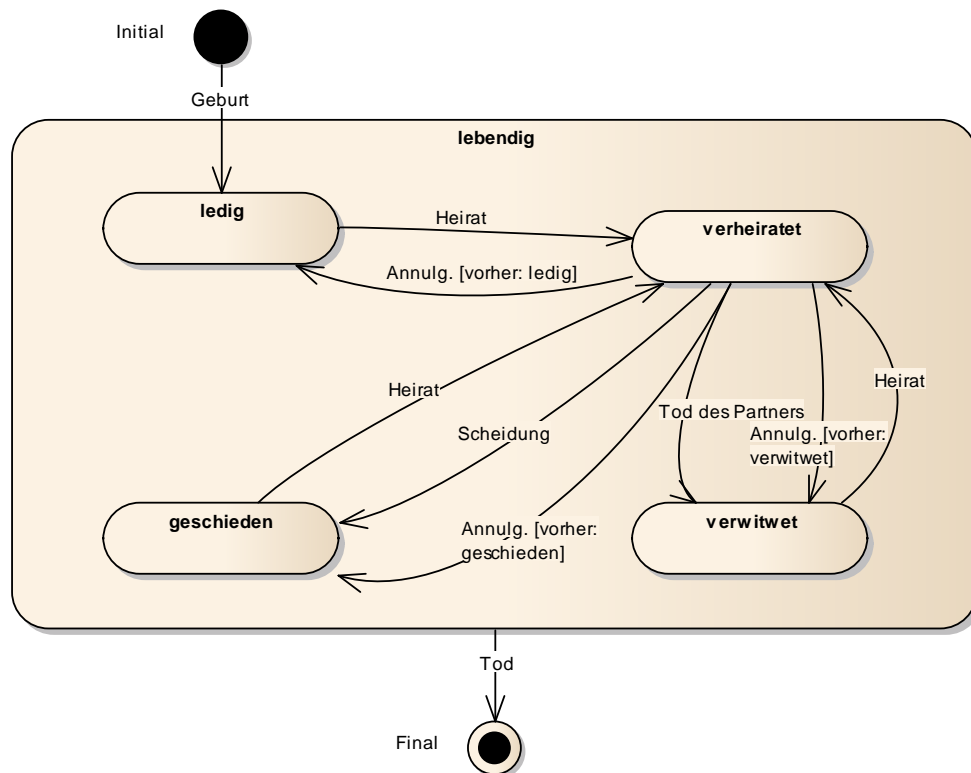


Abb. 28: Beispiel State-Machine-Diagramm

Zustände (States)

Zustände werden durch abgerundete Rechtecke modelliert. Sie können einen Namen beinhalten und optional durch horizontale Linien in bis zu drei Bereiche eingeteilt werden. Im obersten Bereich steht der Name des Zustandes. Wenn der Name nicht angegeben wird, handelt es sich um einen anonymen Zustand. In einem weiteren Bereich können existierende Zustandsvariablen mit der für diesen Zustand typischen Wertebelegung angeführt werden. Der dritte Bereich innerhalb des Zustandssymbols kann eine Liste von internen Ereignissen, Bedingungen und aus ihnen resultierende Operationen enthalten.

Ereignis steht dabei für drei mögliche Verhaltensweisen:

- entry - löst automatisch die angegebene Aktivität beim Eintritt in einen Zustand aus, also bei allen hereinkommenden Übergängen.
- exit - löst automatisch beim Verlassen eines Zustandes aus, also bei allen abgehenden Übergängen.
- do - wird immer wieder ausgelöst, solange der Zustand nicht gewechselt wird.

Zustandsübergänge (Transitions)

Übergänge von einem Zustand zum nächsten werden durch Ereignisse ausgelöst. Ein Ereignis besteht aus einem Namen und einer Liste möglicher Argumente. Ein Zustand kann Bedingungen an dieses Ereignis knüpfen, die erfüllt sein müssen, damit dieser Zustand durch dieses Ereignis eingenommen wird. Diese Bedingungen können unabhängig von einem speziellen Ereignis sein.

Gleichzeitig mit einem Zustandsübergang kann eine Aktion durchgeführt werden. Die Notation einer Transition sieht folgendermaßen aus:

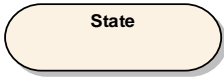


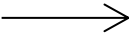
Ereignis [Bedingung] / Action

„Ereignis“, „[Bedingung]“ und „/Action“ sind allesamt optionale Bestandteile. Der Ereigniseintrag am Übergang vom Startpunkt zum ersten State darf fehlen. Auch an anderen Übergängen kann

das Ereignis weggelassen werden. Ist dies der Fall, wird der Zustand automatisch gewechselt, wenn alle Aktivitäten des vorhergehenden Zustandes abgearbeitet wurden (entry..). Das KEIN-Ereignis (Trigger) wird auch als ANY-Trigger bezeichnet, dieses Ereignis ist IMMER vorhanden.

Symbole

Die folgende Tabelle enthält die Symbole der Zustandsdiagramme.

Name/Symbol	Verwendung
Zustand 	Der Zustand eines Objekts wird durch ein Rechteck mit abgerundeten Ecken symbolisiert. Innerhalb dieses Symbols wird der Zustand benannt.
Objekterzeugung 	Der Startpunkt des Zustandsdiagramms wird mit einem gefüllten Kreis dargestellt. Er ist identisch mit der Objekterzeugung. Nur ein Startpunkt pro State-Diagramm ist zulässig und muss vorhanden sein. Die Anordnung des Startpunkts ist freigestellt.
Objektzerstörung 	Die Kette der Zustandsübergänge endet mit der Objektzerstörung. Der Endpunkt wird mit einem gefüllten Kreis dargestellt, den ein konzentrischer Kreis umgibt. Bei endlos laufenden Prozessen kann dieses Symbol in der Zeichnung fehlen, es darf aber auch mehrfach eingetragen werden. Das Token kehrt ggf. an das Ende der Activity im übergeordneten Diagramm, die das Unterdiagramm aufgerufen hat, zurück.
Übergang 	Über einen Pfeil wird der Übergang (Transition) eingezeichnet. Der Pfeil wird mit dem Namen Auslösers (Trigger) beschriftet, die den Objektzustand ändert. Sie können in eckigen Klammern eine Restriktion [Guard] angeben, die bewirkt, dass nur bei deren Erfüllung der Objektzustand geändert wird. Zusätzlich kann hinter „/“ eine Aktivitätenliste, die beim Übergang auszuführen ist, angegeben werden. Guard und Aktivitätenliste sind optional, auch der Trigger (Auslöser) kann fehlen – am Übergang von Initial und wenn ein ANY-Trigger modelliert wird.

Beispiel

Hochlauf eines Bankomaten und Hauptzustände. Beim Einschalten durchläuft der Bankomat einen Selbsttest. Abhängig davon, wie dieser ausgeht, wird die Grundstellung erreicht oder der Störungszustand. Zusätzlich wurde vorgesehen, dass bei überlanger Dauer des Selbsttests auch der Störungszustand eingenommen wird. Wird die Karte eingeschoben, erfolgt die Kartenprüfung. Abhängig von ihrem Ergebnis gelangt das Gerät in die Pin-Abfrage oder in den Abbruchzustand. Die weiteren States wie Betragsabfrage, Verfügbarkeitsprüfung, usw. sind nicht mehr dargestellt.

Die Kettensymbole zeigen an, dass Subdiagramme vorhanden sind, die das Verhalten in den Zuständen näher beschreiben. Es besteht die Freiheit als Subdiagramme beliebige Verhaltensdiagramme zu verwenden, es muss sich nicht unbedingt um weitere Zustandsdiagramme handeln.

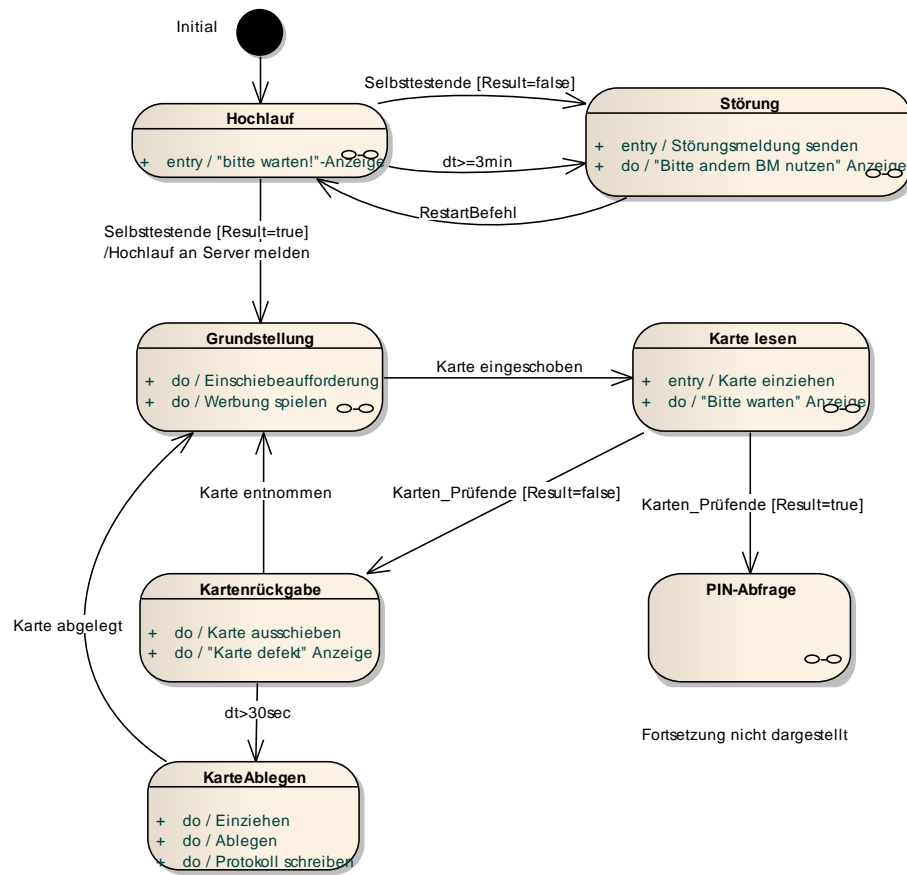


Abb. 29: Beispiel Zustandsdiagramm „Bankomathochlauf“

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Ein Zustand ohne notierten Namen ist

- [a] nicht UML-konform.
- [b] ein unvollständiger Zustand.
- [c] ein „anonymer“ Zustand.

2. Der Startpunkt des State Diagram

- [a] ist das erste eintreffende Ereignis
- [b] definiert einen Guard für den ersten Zustandswechsel.
- [c] ist gleichbedeutend mit der Objekterzeugung.

Richtige Antworten: 1c, 2c

Klassendiagramm (Class Diagram)

Das Klassendiagramm bildet das Herzstück der UML. Es basiert auf den Prinzipien der Objekt-orientierung (Abstraktion, Kapselung, Vererbung, ...) und ist durch seine Vielseitigkeit in allen Phasen eines Projekts einsetzbar. In der Analysephase tritt es als Domainmodell in Erscheinung und versucht ein Abbild der Wirklichkeit darzustellen. In der Designphase werden damit Datenstrukturen und die Software modelliert und in der Implementierungsphase wird daraus Sourcecode generiert.

In Klassendiagrammen werden Klassen und die Beziehungen von Klassen untereinander modelliert. Bei den Beziehungen kann man grob drei Arten unterscheiden. Die einfachste und allgemeinste Variante ist die Assoziation. Eine zweite modellierbare Beziehung ist die Aufnahme einer Klasse in eine zweite Klasse, die sogenannte Containerklasse. Solche Beziehungen werden Aggregation oder Komposition genannt. Eine dritte Möglichkeit ist die Spezialisierung bzw. Generalisierung.

Da eine Klasse die Struktur und das Verhalten von Objekten, die von dieser Klasse erzeugt werden, modellieren muss, können diese mit Methoden und Attributen versehen werden. Weiterhin ist die Modellierung von Basisklassen und Schnittstellen über Stereotype möglich. In einigen Programmiersprachen können Templateklassen implementiert werden. Die UML stellt solche Klassen im Klassendiagramm als parametrisierbare Klassen dar.

Klasse

Eine Klasse (Class) beschreibt eine Menge von Instanzen, die dieselben Merkmale, Zusicherungen und Semantik haben.

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse tragen oder zusätzlich auch Attribute und Operationen. Dabei werden diese drei Rubriken (Compartments) - Klassenname, Attribute, Operationen - jeweils durch eine horizontale Linie getrennt. Klassennamen beginnen gewöhnlich mit einem Großbuchstaben und sind meist Substantive im Singular (Sammlungsklassen u. Ä. ggf. im Plural).

Die Attribute einer Klasse werden mindestens mit ihrem Namen aufgeführt und können zusätzlich Angaben zu ihrem Typ, einen Initialwert, Eigenschaftswerte und Zusicherungen enthalten. Methoden werden ebenfalls mindestens mit ihrem Namen, zusätzlich mit möglichen Parametern, deren Typ und den Initialwerten, sowie eventueller Eigenschaftswerte und Zusicherungen notiert.

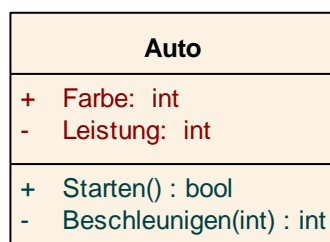


Abb. 30: Beispiel Klasse

Sichtbarkeitsbereich

Der Sichtbarkeitsbereich (Scope) der Klassenelemente wird mit einem Zeichen vor dem Namen gekennzeichnet. Ist ein Element öffentlich sichtbar, steht vor dem Namen das Zeichen +. Private Elemente erhalten das Zeichen - vorangestellt. Das Zeichen # vor dem Namen bedeutet, dass das Klassenelement mit dem Zugriffsattribut protected gekennzeichnet ist. Protected bedeutet eine Erweiterung von private – Tochterklassen erben Attribute, die als protected gekennzeichnet sind. ~ vor dem Namen bedeutet Package – eine Einschränkung von public; nicht unbegrenzte öffentliche Sichtbarkeit, sondern begrenzt auf das Package.

Abstrakte Klasse

Von einer abstrakten Klasse werden niemals Instanzen erzeugt. Sie ist bewusst unvollständig und bildet somit die Basis für weitere Unterklassen, die Instanzen haben können. Eine abstrakte Klasse wird wie eine normale Klasse dargestellt, jedoch wird der Klassenname kursiv gesetzt.

Stereotype

Durch Stereotype können z. B. abstrakte Klassen gekennzeichnet werden. Die Angabe des Stereotyps erscheint ober dem Klassennamen in französischen Anführungsstrichen « ». Stereotype können auch durch unterschiedliche Farben oder durch die Kursivschreibweise des Namens der Klasse sichtbar gemacht werden.

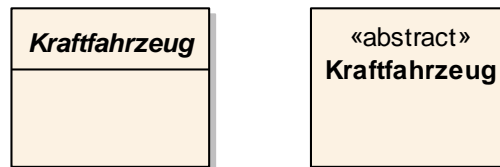


Abb. 31: Beispiel Stereotype

Parametrisierbare Klassen

Eine besondere Art von Klassen sind parametrisierbare Klassen. Bei diesen Klassen ist der Typ der enthaltenen Attribute noch nicht festgelegt. Die Festlegung erfolgt erst beim Instanzieren eines Objekts dieser Klassen. Für solche Klassen wird die grafische Darstellung abgewandelt. Das Rechteck für die Klasse bekommt im oberen Bereich ein zweites Rechteck mit einer Umrandung, in dem der variierbare Typ steht.

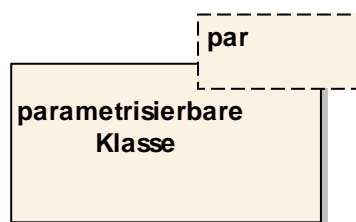


Abb. 32: Parametrisierbare Klasse

Objekt

Objekte sind die konkreten, agierenden Einheiten einer objektorientierten Anwendung. Sie werden nach einem Bauplan, der Klassendefinition, im Speicher erzeugt. Jedes Objekt hat eine eigene Identität. Ein Objekt besitzt ein bestimmtes Verhalten, das durch seine Methoden bestimmt ist. Zwei Objekte der gleichen Klasse haben das gleiche Verhalten. Objekte besitzen weitere Attribute, die bei Objekten derselben Klasse gleich sind. Der Zustand eines Objekts wird durch die Werte bestimmt, die in den Attributen gespeichert sind. Deshalb sind zwei Objekte einer Klasse gleich, wenn die Werte in den Attributen übereinstimmen.

Im Diagramm werden Objekte analog zu den Klassen mit einem Rechteck gezeichnet. Der Name wird unterstrichen, um sie von den Klassen unterscheiden zu können. Dem Namen des Objekts wird nach einem Doppelpunkt der Klassenname angefügt. Hat für den zu modellierenden Fall der konkrete Objektname keine Bedeutung, kann dieser auch entfallen. Es werden dann nur ein Doppelpunkt und der Klassenname dargestellt. Da die Methoden in der Objektdarstellung uninteressant sind, werden diese nicht angegeben.



Abb. 33: Beispiel Objekte

Eigenschaften (Attribute)

Ein Attribut ist ein (Daten-)Element, das in jedem Objekt einer Klasse gleichermaßen enthalten ist und von jedem Objekt mit einem individuellen Wert repräsentiert wird.

Anders als in der UML 1.x wird in der UML 2.0 und danach nicht mehr strikt zwischen Attributen und Assoziationsenden unterschieden. Das heißt, die Darstellung als Attribut in einer Klasse oder als navigierbare Assoziation ist gleichbedeutend.

Jedes Attribut wird mindestens durch seinen Namen beschrieben. Zusätzlich können ein Typ, die Sichtbarkeit, ein Initialwert u. Ä. definiert werden. Die vollständige Syntax lautet, wie folgt

[Sichtbarkeit][[/]Name[:Typ][Multiplizität][=Initialwert]

Methoden (Operationen)

Eine Klasse muss für jede Nachricht, die sie empfängt, eine zuständige Methode besitzen. Über eine Methode stellt eine Klasse anderen Klassen Funktionalität zur Verfügung. Über Nachrichten, sprich Methodenaufrufe, kommunizieren die aus den Klassen installierten Objekte untereinander und erreichen so eine Koordinierung ihres Verhaltens. Objekte und ihre Kommunikation über Methodenaufrufe werden in der Gruppe der Interaktionsdiagramme dargestellt.

Beziehungen

Es gibt vier verschiedene Arten von Beziehungen zwischen Klassen, wobei die Generalisierung eine Sonderform ist, die anderen drei, Assoziation, Aggregation und Komposition sind einander sehr ähnlich sind.

Assoziation

Die Assoziation stellt die Kommunikation zwischen zwei Klassen im Diagramm dar. Die Klassen werden mit einer einfachen Linie verbunden. Mithilfe eines Pfeils kann man eine gerichtete Assoziation modellieren.

Jede Assoziation kann mit einem Namen versehen werden, der die Beziehung näher beschreibt. Der Name kann zusätzlich mit einer Leserichtung - einem kleinen ausgefüllten Dreieck - versehen werden. Diese Richtung bezieht sich nur auf den Namen und hat keinen Bezug zur Navigierbarkeit der Assoziation.

Auf jeder Seite der Assoziation können Rollennamen dazu verwendet werden, genauer zu beschreiben, welche Rolle die jeweiligen Objekte in der Beziehung einnehmen. Die Rollen sind die Namen der Eigenschaften (Attribute), die der Assoziation oder einer der beteiligten Klassen gehören.

Assoziationen werden in Programmiersprachen in der Regel dadurch realisiert, dass die beteiligten Klassen entsprechende Attribute erhalten.

Eine Rolle repräsentiert also eine Eigenschaft. Außer Rollennamen können auch Sichtbarkeitsangaben auf jeder Seite der Assoziation angebracht werden. Ist beispielsweise ein Assoziationsende als privat (-) deklariert, so kann das Objekt selbst, d. h. die Operationen des Objekts, die Assoziation benutzen, benachbarte Klassen erhalten jedoch keinen Zugriff.

Eine gerichtete Assoziation wird wie eine gewöhnliche Assoziation notiert, jedoch hat sie auf der Seite der Klasse, zu der navigiert werden kann, also in Navigationsrichtung, eine geöffnete Pfeil-

spitze. Multiplizität und Rollename können theoretisch jedoch auch auf der Seite notiert werden, zu der nicht navigiert werden kann. Sie bezeichnen dann eine Eigenschaft (*Property*), die zu keiner Klasse gehört, sondern zur Assoziation.

In dem folgenden Beispiel würde die Klasse „Kunde“ ein Attribut "konto" als Referenz auf ein Objekt der Klasse Kundenkonto erhalten und die Klasse Kundenkonto ein privates Attribut "bpos" mit einem Sammlungsobjekt (Collection bzw. Unterklasse davon), welches die Buchungsposition-Objekte referenziert.



Abb. 34: Assoziation und Komposition mit allen Angaben

Viele Modellierungswerkzeuge verwenden die Rollennamen der Beziehung für die entsprechenden automatisch erzeugten Attribute, Rollennamen korrespondieren in der UML auch formal mit den entsprechenden Attributen. Assoziation mit Navigierbarkeit ist eine Alternativnotation zur Attributdarstellung in der zugehörigen Klasse.

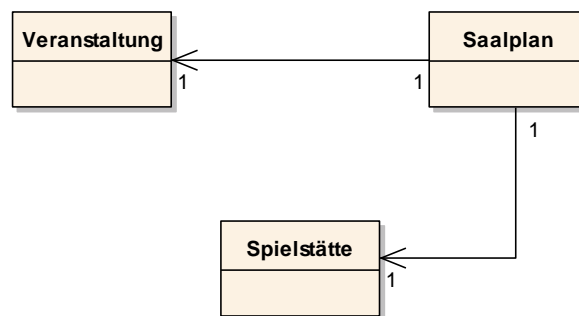


Abb. 35: Assoziationen

Gelesen werden diese Beziehungen in folgender Weise:

- Eine Veranstaltung hat einen Saalplan.
- Ein Saalplan ist einer Spielstätte zugeordnet.

Der Pfeil sagt aus, dass die Kommunikation überwiegend vom Saalplan ausgeht (bei der Implementierung erhält deshalb die Klasse eine Referenz auf die Spielstätte). Dabei werden die Kardinalitäten vor der Zielklasse gelesen.

Multiplizität

In der UML gibt es den Begriff Multiplizität, um die Menge möglicher Ausprägungen zu beschreiben. Die Multiplizität wird durch einen minimalen Wert und einen maximalen Wert beschrieben, z. B. 3..7. Ist der minimale Wert gleich dem maximalen Wert kann die Unter- oder Obergrenze entfallen. Es wird dann anstelle von 3..3 lediglich 3 geschrieben. Ist die untere Grenze 0, so bedeutet dies, dass die Beziehung optional ist.

In der UML wird auch der Begriff Kardinalität verwendet, um die Anzahl konkreter Ausprägungen zu beschreiben.

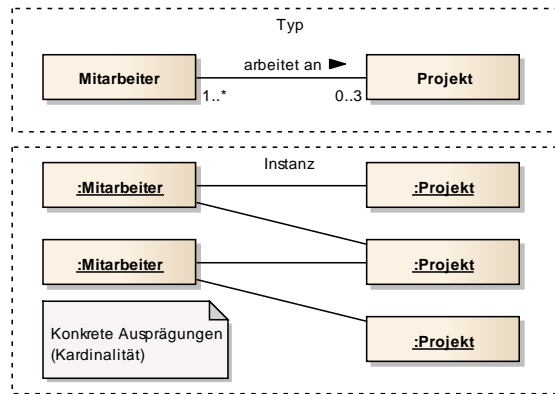


Abb. 36: Multiplizität vs. Kardinalität

Der Begriff Kardinalität hat seinen Ursprung in der Datenmodellierung und dort die gleiche Bedeutung wie die Multiplizität in der UML. In der UML werden beide Begriffe verwendet und somit feiner zwischen möglichen Ausprägungen und tatsächlichen Ausprägungen unterschieden. Die Kardinalität beschreibt z. B. beim Objektmodell die konkrete Anzahl assoziierter Objekte.

Assoziationsklasse

Klassen können aber auch kombinatorisch untereinander in Beziehung stehen; betrachtet man die Beziehungen zwischen Kunde, Konto und Bankomatkarte, dann stellt sich heraus, dass eine Bankomatkarte für eine Kombination aus einem Kunden und einem Konto existiert und das ist etwas gänzlich anderes, als die Karte mit Kunde und Konto direkt zu assoziieren! Die UML stellt dafür die Schreibweise Assoziationsklasse zur Verfügung:

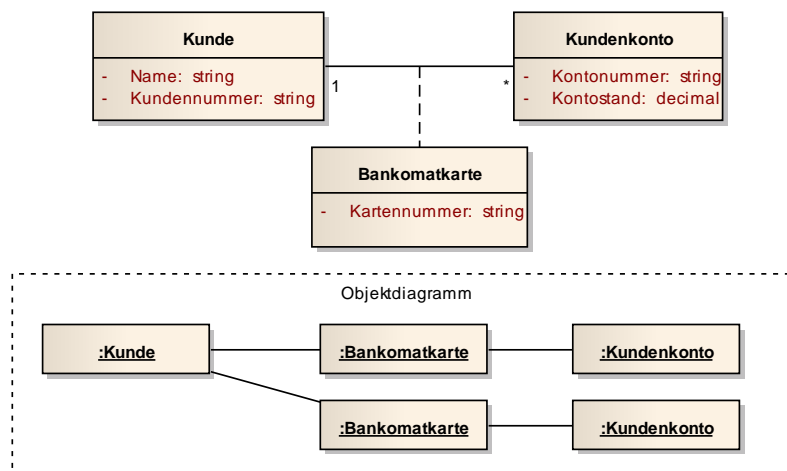


Abb. 37: Assoziationsklasse

Eine Besonderheit der Assoziationsklasse ist, dass es nur genau eine Instanz der assoziierten Klasse pro Referenz geben darf. Im Beispiel darf es also nur eine Instanz von Bankomatkarte für eine Kombination aus Kunde und Konto geben.

Sind mehrere Klassen beteiligt, dann wird der Assoziationsknoten (n-äre Assoziation) verwendet:

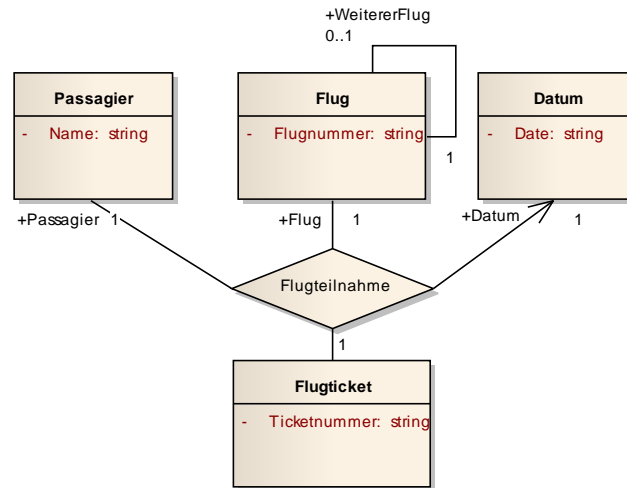


Abb. 38: Assoziationsknoten (n-äre Assoziation)

Die n-äre Assoziation hat ihren Ursprung in der Datenmodellierung (Entity Relationship Modelle) und wird in der UML mit identischer Semantik verwendet. Alle an der Assoziation beteiligten Klassen bilden eine Einheit. Im Beispiel bedeutet dies, dass (Flugticket, Datum, Flug, Passagier) eine Einheit bilden und als Gesamtes eine Bedeutung haben, die Teilnahme an einem konkreten Flug. Die Bedeutung wird meist im Namen der Assoziation (Raute) ausgedrückt.

Wichtig bei der n-ären Assoziation ist das setzen der Multiplizitäten, d. h. welche Kombinationen von Objektausprägungen sind gültig. Um die Multiplizitäten richtig setzen zu können, denken Sie für n-1 Elemente eine Multiplizität von 1 und setzen die Multiplizität des n-ten.

Zum Beispiel: Ein Passagier hat für einen Flug an einem Datum genau ein Flugticket. Falls es möglich sein sollte, dass der Passagier mehrere Tickets für denselben Flug am gleichen Datum besitzen darf, muss die Multiplizität bei Flugticket größer als 1 sein (z. B. *).

Aggregation

Eine Aggregation ist eine Assoziation, erweitert um den semantisch unverbindlichen Kommentar, dass die beteiligten Klassen keine gleichwertige Beziehung führen, sondern eine "Teil von"-Beziehung darstellen. Eine Aggregation soll beschreiben, wie sich etwas Ganzes aus seinen Teilen logisch zusammensetzt.

Eine Aggregation wird wie eine Assoziation als Linie zwischen zwei Klassen dargestellt und zusätzlich mit einer kleinen Raute versehen. Die Raute steht auf der Seite des Aggregats, also des Ganzen. Sie symbolisiert gewissermaßen das Behälterobjekt, in dem die Einzelteile gesammelt sind. Im Übrigen gelten alle Notationskonventionen der Assoziation.

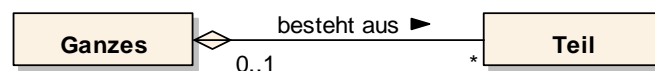


Abb. 39: Notation Aggregation

Das Beispiel Unternehmen-Abteilung-Mitarbeiter zeigt, dass ein Teil (Abteilung) gleichzeitig auch wieder Aggregat sein kann.

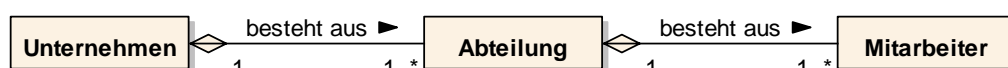


Abb. 40: Beispiel Aggregation

Komposition

Eine Komposition ist eine strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind. Sie beschreibt, wie sich etwas Ganzes aus Einzelteilen zusammensetzt. Die Kom-

position wird wie die Aggregation als Linie zwischen zwei Klassen gezeichnet und mit einer kleinen Raute auf der Seite des Ganzen versehen. Im Gegensatz zur Aggregation wird die Raute jedoch ausgefüllt.

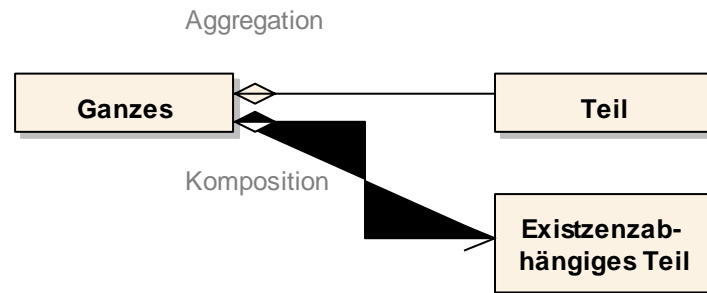


Abb. 41: Aggregation und Komposition

Wenn eine Klasse als Teil mehrerer Kompositionen definiert wird, wie beispielsweise in der folgenden Abbildung, dann bedeutet dies, dass sowohl Pkws als auch Boote einen Motor besitzen, ein konkretes Motor-Objekt aber immer nur entweder zu einem Pkw oder einem Boot gehören kann. Das entscheidende Merkmal einer Komposition ist, dass die aggregierten Teile niemals mit anderen Objekten geteilt werden. Dies kann auch im Diagramm mittels XOR-Constraint beschrieben werden.

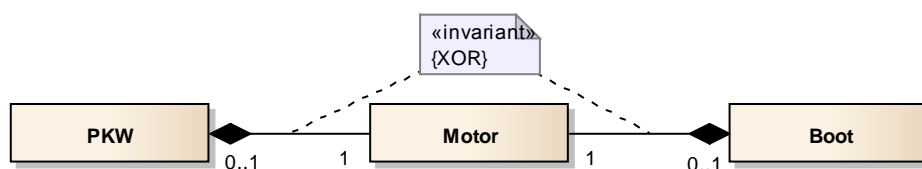


Abb. 42: Beispiel Komposition

Falls die Multiplizität für Teile (Motor) von null beginnt (0..*), dann kann das Ganze auch ohne Teile existieren, bzw. beliebig viele Teile beinhalten. Falls die Multiplizität des Ganzen (PKW/Boot) null bis eins (0..1) beträgt, dann kann das Teil auch ohne Ganzem existieren, sobald jedoch das Teil zum Ganzen gehört, wird es nicht mehr getrennt und es entsteht eine existenzabhängige Beziehung. Existenzabhängig bedeutet, dass das Teil ohne seinem Ganzen nicht existieren kann und wenn das Ganze zerstört wird, müssen, auch alle seine Teile zerstört werden.

In der folgenden Abbildung ist zwischen der Klasse *Veranstaltungsplan* und der Klasse *Saalplan* eine Aggregation modelliert. Eine Komposition ist zwischen den Klassen *Saalplan* und *Platz* definiert. Es gibt somit kein Saalplan-Objekt ohne einen Platz. Gelesen werden die Beziehungen in folgender Weise:

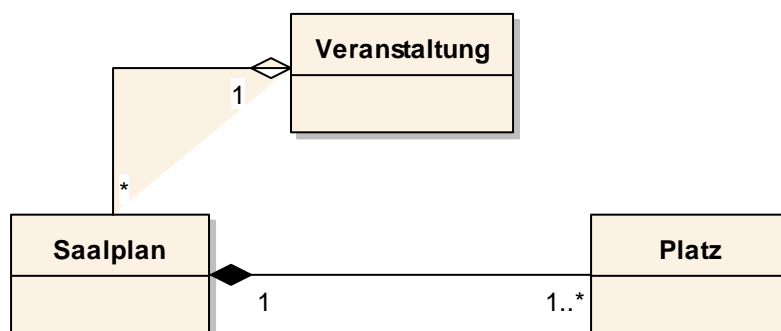


Abb. 43: Beispiel Aggregation und Komposition

- Ein Saalplan hat beliebig viele Plätze, aber mindestens einen.
- Ein Platz gehört genau zu einem Saalplan.

- Jeder Saalplan gehört zu einer Veranstaltung.
- Eine Veranstaltung kann beliebig viele Saalpläne enthalten.
- Der Saalplan könnte auch zu einer anderen Veranstaltung zugeordnet werden (Aggregation), muss allerdings immer eine Veranstaltung haben.
- Der Platz gehört zu einem Saalplan, diese Beziehung kann nicht geändert werden (Komposition).

Generalisierung/Spezialisierung

Eine Generalisierung ist eine Beziehung zwischen einer allgemeinen und einer speziellen Klasse, wobei die speziellere weitere Merkmale hinzufügen kann und sich kompatibel zur allgemeinen verhält.

Bei der Generalisierung bzw. Spezialisierung werden Merkmale hierarchisch gegliedert, d. h., Merkmale allgemeinerer Bedeutung werden allgemeineren Elementen zugeordnet und spezielle Merkmale werden Elementen zugeordnet, die den allgemeineren untergeordnet sind. Die Merkmale der Ober-Elemente werden an die entsprechenden Unter-Elemente weitergegeben, d. h. vererbt. Ein Unter-Element verfügt demnach über die in ihm spezifizierten Merkmale sowie über die Merkmale seiner Ober-Elemente. Unter-Elemente erben alle Merkmale ihrer Ober-Elemente und können diese um weitere erweitern oder überschreiben.

Mit dieser Form werden hierarchische Vererbungsbäume modelliert. Vererbungsbäume sind ein wichtiges Gestaltungselement bei der Modellierung von Softwarearchitekturen. In der folgenden Abbildung wird beispielsweise modelliert, dass Veranstaltungen Inhouse- oder Gastveranstaltungen sein können.

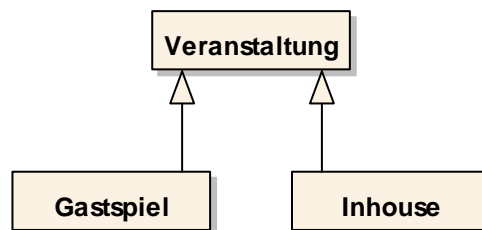


Abb. 44: Beispiel Vererbung

Abhängigkeiten (Dependencies)

Eine Abhängigkeit (Dependency) ist eine Beziehung von einem (oder mehreren) Quellelement(en) zu einem (oder mehreren) Zielelement(en).

Die Zielelemente sind für die Spezifikation oder Implementierung der Quellelemente erforderlich. Dargestellt wird eine Abhängigkeit durch einen gestrichelten Pfeil, wobei der Pfeil vom abhängigen auf das unabhängige Element zeigt. Zusätzlich kann die Art der Abhängigkeit durch ein Schlüsselwort oder Stereotyp näher spezifiziert werden.

Da das Quellelement bei einer Abhängigkeitsbeziehung das Zielelement für seine Spezifikation oder Implementierung benötigt, ist es ohne das Element unvollständig.

Abhängigkeiten können unterschiedliche Ursachen haben. Einige Beispiele hierfür sind:

- Ein Paket ist von einem anderen abhängig. Die Ursache kann hierbei beispielsweise darin bestehen, dass eine Klasse in dem einen Paket von einer Klasse in dem anderen Paket abhängig ist.
- Eine Klasse benutzt eine bestimmte Schnittstelle einer anderen Klasse. Wenn die angebotene Schnittstelle verändert wird, sind in der schnittstellennutzenden Klasse ebenfalls Änderungen erforderlich.

- Eine Operation ist abhängig von einer Klasse, beispielsweise wird die Klasse in einem Operationsparameter genutzt. Eine Änderung in der Klasse des Parameters macht möglicherweise eine Änderung der Operation erforderlich.

Stereotype	Bedeutung
«call»	Stereotyp an Verwendungsbeziehung (Usage) Die call-Beziehung wird von einer Operation zu einer anderen Operation definiert und spezifiziert, dass die Quelloperation die Zieloperation aufruft. Als Quellelement kann auch eine Klasse gewählt werden. Das bedeutet dann, dass die Klasse eine Operation enthält, die die Zieloperation aufruft.
«create»	Stereotyp an Verwendungsbeziehung (Usage) Das abhängige Element erzeugt Exemplare des unabhängigen Elementes. Die create-Beziehung wird zwischen Klassen definiert.
«derive»	Stereotyp an Abstraktionsbeziehung (Abstraction) Das abhängige Element ist vom unabhängigen Element abgeleitet.
«instantiate»	Stereotyp an Verwendungsbeziehung (Usage) Das abhängige Element erzeugt Exemplare des unabhängigen Elementes. Die Beziehung wird zwischen Klassen definiert.
«permit»	Schlüsselwort für Berechtigungsbeziehung (Permission) Das abhängige Element hat die Erlaubnis, private Eigenschaften des unabhängigen Elementes zu verwenden.
«realize»	Schlüsselwort für Realisierungsbeziehung (Realization) Das abhängige Element implementiert das unabhängige Element, beispielsweise eine Schnittstelle oder ein abstraktes Element oder ein Element muss ein Requirement umsetzen.
«refine»	Stereotyp an Abstraktionsbeziehung (Abstraction) Das abhängige Element befindet sich auf einem konkreteren semantischen Niveau als das unabhängige Element.
«trace»	Stereotyp an Abstraktionsbeziehung (Abstraction) Das abhängige Element führt zum unabhängigen Element, um semantische Abhängigkeiten nachverfolgen zu können, beispielsweise von einer Klasse zu einer Anforderung oder von einem Anwendungsfall zu einer Klasse. Häufig auch verwendet zwischen Testfall und betestetem Element.
«use»	Schlüsselwort für Verwendungsbeziehung (Usage) Das abhängige Element benutzt das unabhängige Element für seine Implementierung.

Die Abstraktionsbeziehung (Abstraction) ist eine spezielle Abhängigkeitsbeziehung zwischen Modellelementen auf verschiedenen Abstraktionsebenen. Die Beziehung wird wie eine Abhängigkeitsbeziehung notiert. Sie können aber nicht verwechselt werden, da die Abstraktion immer zusammen mit einem Stereotyp verwendet wird. Die UML definiert die Standard-Stereotype «derive», «trace» und «refine».

Zu einer Abstraktionsbeziehung gehört auch immer eine Abbildungsvorschrift, wie die beteiligten Elemente zueinanderstehen (Mapping). Die Angabe kann formal oder informal sein. Eine Abstraktionsbeziehung kann trotz der Pfeilrichtung je nach Stereotyp und Abbildungsvorschrift auch bidirektional sein, z. B. häufig die «trace»-Beziehung.

Eine spezielle Abstraktionsbeziehung ist die *Realisierungsbeziehung (Realization)*. Es ist eine Beziehung zwischen einer Implementierung und ihrem Spezifikationselement.

Die *Substitutionsbeziehung (Substitution)* ist eine spezielle Realisierungsbeziehung. Sie gibt an, dass Exemplare des unabhängigen Elementes zur Laufzeit durch Exemplare des abhängigen Elementes substituiert werden können, z. B. da die Elemente dieselben Schnittstellen implementieren.

Die *Verwendungsbeziehung* (*Usage*) ist eine spezielle Abhängigkeitsbeziehung. Der Unterschied zur Abhängigkeitsbeziehung ist, dass die Abhängigkeit sich nur auf die Implementierung beschränkt und nicht für die Spezifikation gilt. Das heißt, das abhängige Element benötigt das unabhängige Element für seine Implementierung. Es kann also keine Verwendungsbeziehungen zwischen Schnittstellen geben, aber dafür die Abhängigkeitsbeziehung.

Die *Berechtigungsbeziehung* (*Permission*) ist eine spezielle Abhängigkeitsbeziehung, die dem abhängigen Element Zugriffsrechte auf das unabhängige Element erteilt.

Schnittstellen

Schnittstellen (Interfaces) sind spezielle Klassen, die mit einer Menge von Merkmalen (Features) einen ausgewählten Teil des extern sichtbaren Verhaltens von Modellelementen (hauptsächlich von Klassen und Komponenten) spezifizieren.

Die Implementierungsbeziehung (Implementation) ist eine spezielle Realisierungsbeziehung zwischen einer Klasse und einer Schnittstelle. Die Klasse implementiert die in der Schnittstelle spezifizierten Merkmale.

Schnittstellen werden wie Klassen notiert, sie tragen jedoch das Schlüsselwort «interface».

Es werden bereitgestellte und benötigte Schnittstellen unterschieden. Eine bereitgestellte Schnittstelle wird von einem Modellelement angeboten und kann von anderen verwendet werden. Eine benötigte Schnittstelle ist eine Schnittstelle, die von einem anderen Modellelement eingefordert wird.

Schnittstellen spezifizieren nicht nur Operationen, sondern auch Attribute. Entsprechend dürfen Schnittstellen auch Assoziationen zu anderen Schnittstellen oder Klassen haben.

Klassen, die eine Schnittstelle implementieren wollen, müssen alle in der zugehörigen Schnittstelle definierten Operationen implementieren. Bezüglich der in der Schnittstelle definierten Attribute muss sich die implementierende Klasse so verhalten, als ob sie die Attribute besitzt. Im einfachsten Fall implementiert sie die Attribute wirklich. Es genügt aber auch, z. B. nur die entsprechenden `get()` und `set()` Methoden anzubieten.

Eine Klasse kann mehrere Schnittstellen implementieren und darüber hinaus weitere Eigenschaften enthalten oder anders ausgedrückt: Eine Schnittstelle beschreibt in der Regel eine Untermenge der Operationen und Attribute einer Klasse.

Zwischen implementierender Klasse und der Schnittstelle besteht eine spezielle Realisierungsbeziehung, die Implementierungsbeziehung (Schlüsselwort «realize»).

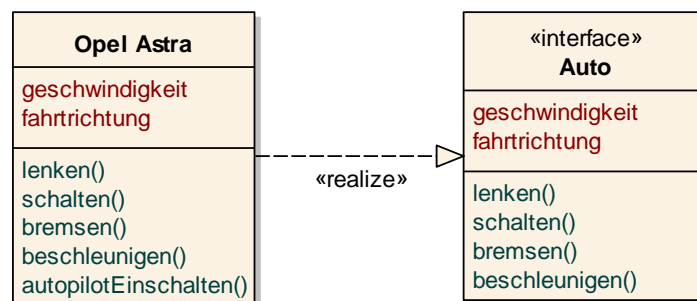


Abb. 45: Beispiel Schnittstelle

Aus der aktuellen Spezifikation geht nicht klar hervor, ob für die Implementierungsbeziehung der gestrichelte Pfeil mit Schlüsselwort «realize» oder wie in der UML 1.x die gestrichelte Generalisierungsbeziehung verwendet wird. Wir nehmen hier Letzteres an, da diese Variante auch für die praktische Arbeit wesentlich besser geeignet ist.

Die Implementierungsbeziehung wird mit einem speziellen Pfeil notiert. Die andere Möglichkeit, die Implementierung einer Schnittstelle darzustellen, ist ein kleiner, nicht ausgefüllter Kreis, der durch eine Linie mit der Klasse verbunden ist, der die Schnittstelle anbietet. Dies soll einen Stecker symbolisieren. Daneben wird der Name der Schnittstelle genannt; er entspricht dem Namen der zugehörigen Schnittstelle.

Bei der Pfeilschreibweise besteht die Möglichkeit, die durch die Schnittstelle geforderten Operationen und Attribute abzulesen. Bei der Kurznotation sieht man die von der Schnittstelle geforderten Operationen und Attribute nicht, sondern nur den Namen der Schnittstelle.

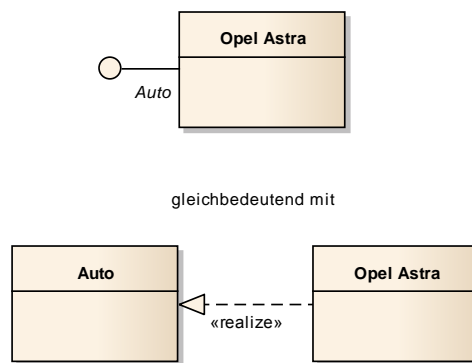


Abb. 46: Notation bereitgestellte Schnittstelle

Für eine angeforderte Schnittstelle besitzt das anfordernde Element eine Verwendungsbeziehung zu der Schnittstelle. Bei der dafür existierenden Kurznotation handelt es sich um einen Halbkreis am Stiel. Dies soll eine Buchse symbolisieren.

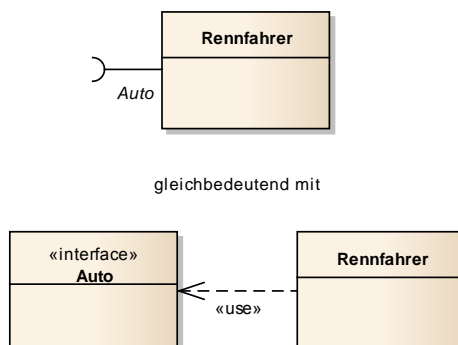


Abb. 47: Notation angeforderte Schnittstelle

Bereitstellung und Anforderung einer Schnittstelle können ebenso durch die Kombination der beiden Kurznotationen dargestellt werden, indem man den Stecker in die Buchse steckt.

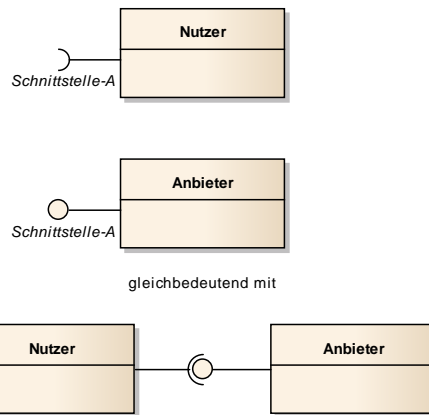


Abb. 48: Notationsmöglichkeiten für Schnittstellen (Nutzung und Bereitstellung)

Vor der letzten Darstellung sei allerdings gewarnt: Die „Assembly“ gehört im Gegensatz zu den Symbolen für bereitgestellte Schnittstelle und benutzte Schnittstelle in die Kategorie Konnektor! Ein durchgängiges Modellieren mit einem Instance-Classifer-Verweis auf die Schnittstellen-Beschreibung wird NICHT möglich sein. Vermeiden Sie diese Schreibweise!

Da Schnittstellen spezielle Klassen sind, gelten alle entsprechenden Regeln. Beispielsweise ist Generalisierung möglich.

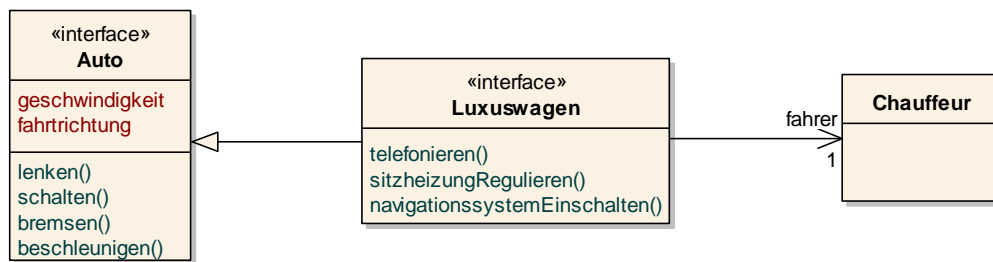


Abb. 49: Erweiterung von Schnittstellen

Die folgende Abbildung zeigt, dass die Klasse „Opel Astra“ die Schnittstelle „Auto“ implementiert. Die Schnittstelle Auto fordert vier Operationen `lenken()`, `schalten()`, `bremsen()` und `beschleunigen()` sowie zwei Attribute „geschwindigkeit“ und „fahrtrichtung“. Die Klasse „Opel Astra“ erfüllt die Anforderungen für diese Schnittstelle, da sie unter anderem über die vier geforderten Operationen und zwei Attribute verfügt.

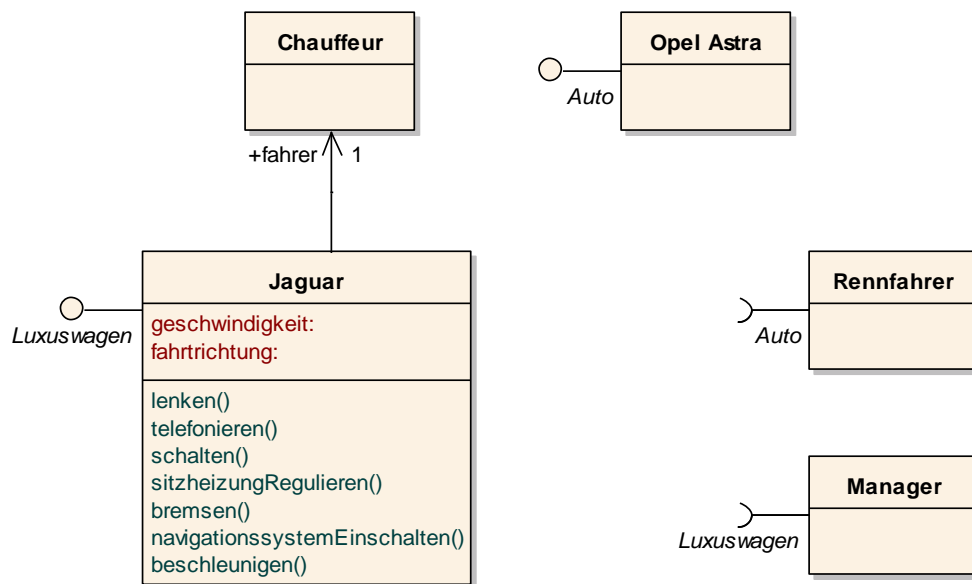
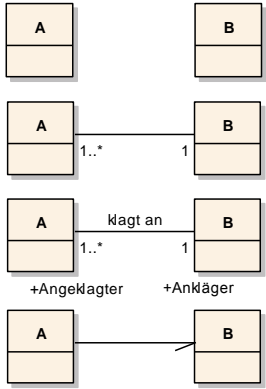
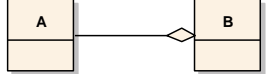
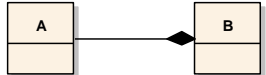
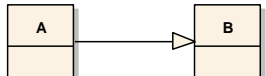
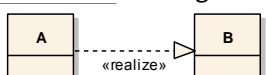


Abb. 50: Beispiel zur Implementierung von Schnittstellen

Symbole

Die folgende Tabelle enthält die Symbole der Klassendiagramme.

Name/Symbol	Verwendung
<p>Klasse</p> <pre> classDiagram class Class { } </pre> <p>Class</p> <pre> classDiagram class Class { - Att1: int + Op1(): void } </pre>	<p>Eine Klasse wird im Diagramm mit einem Rechteck dargestellt, in dem der Name der Klasse angezeigt wird. Das Klassensymbol kann mit einem Bereich zur Darstellung der Attribute und der Methoden erweitert werden. Für parametrierbare Klassen wird das Klassensymbol mit einem Rechteck erweitert, das den Namen des Parameters enthält.</p>
<p>Objekt</p> <pre> classDiagram class Object { } </pre> <p>Object</p> <pre> classDiagram class Class { } </pre> <p>:Class</p> <pre> classDiagram class Object { } </pre> <p>Object :Class</p>	<p>Objekte werden mit einem Rechteck dargestellt, in dem der Name und die Klasse des Objekts unterstrichen angezeigt werden. Beide werden durch einen Doppelpunkt getrennt. Auf den Klassennamen kann verzichtet werden, wenn das Objekt auch ohne Klassennamen eingeordnet werden kann. Ein anonymes Objekt (ohne Namen) wird nur mit einem Doppelpunkt und dem Klassennamen angegeben.</p>
<p>Paket</p> <pre> classDiagram package Package { } </pre> <p>Package</p>	<p>Pakete beinhalten mehrere Klassen, die für eine bestimmte Aufgabe dort gruppiert wurden. Die Grafik für ein Paket symbolisiert eine Karteikarte, auf der der Paketname steht.</p>

Name/Symbol	Verwendung
Assoziation 	<p>Stehen zwei Klassen in einer Beziehung, wird das durch die Verbindung der beiden Klassen mit einer durchgezogenen Linie dargestellt.</p> <p>Auf der Linie kann in der Nähe des Klassensymbols die Kardinalität angegeben werden.</p> <p>Auf der Linie können weiterhin der Name der Beziehung und die Rolle angegeben werden (Rollen: Angeklagter, Ankläger; Beziehungsname: klagt an).</p> <p>Sie können mit einem Pfeil die vorrangige Richtung der Kommunikation angeben. (Navigation)</p>
Aggregation 	<p>Die Aggregation wird mit einer Linie, an deren einem Ende ein Rautensymbol zur Klasse des Containers zeigt, dargestellt (Klasse B kann Objekte der Klasse A verwalten). Das Rautensymbol ist nicht gefüllt. Rollen, Kardinalität und vorrangige Richtung der Kommunikation werden wie bei der Assoziation angegeben.</p>
Komposition 	<p>Die Komposition wird im Unterschied zur Aggregation mit einer gefüllten Raute dargestellt. Die weitere Symbolik entspricht der der Aggregation.</p>
Generalisierung 	<p>Das Symbol ist ein Pfeil, der auf die Basisklasse zeigt und diese mit der abgeleiteten Klasse verbindet. Der Pfeil wird nicht gefüllt dargestellt (Klasse A erbt von der Klasse B).</p>
Realisierung 	<p>Importiert eine Klasse eine Schnittstelle (Interface), wird die Verbindung mit einem Pfeil und einer unterbrochenen Linie dargestellt.</p>

Beispiel

Bei einem Ticketsystem für ein Schauspielhaus werden über einen Veranstaltungsplan alle Stücke des Theaters verwaltet. Die Klasse Veranstaltungsplan ist mit der Klasse Saalplan über eine Komposition verbunden. Somit kann die Klasse Veranstaltungsplan Objekte der Klasse *Saalplan* verwalten. Jeder *Saalplan* enthält die Plätze des Theaters, und diese werden über den *Saalplan* verkauft. Die Klasse *Saalplan* verwaltet über eine weitere Komposition Objekte der Klasse *Platz*. Die Navigation ist in Richtung der Klasse *Platz* gerichtet. Die Klasse *Saalplan* kommuniziert mit der Klasse *Veranstaltung*, in die die Daten der Aufführung ausgelagert sind, z. B. Datum und Uhrzeit. Die Angaben zur Spielstätte, z. B. der Name und die Adresse, wurden in die Klasse *Spielstätte* ausgelagert. Die Navigation erfolgt von der Klasse *Saalplan*.

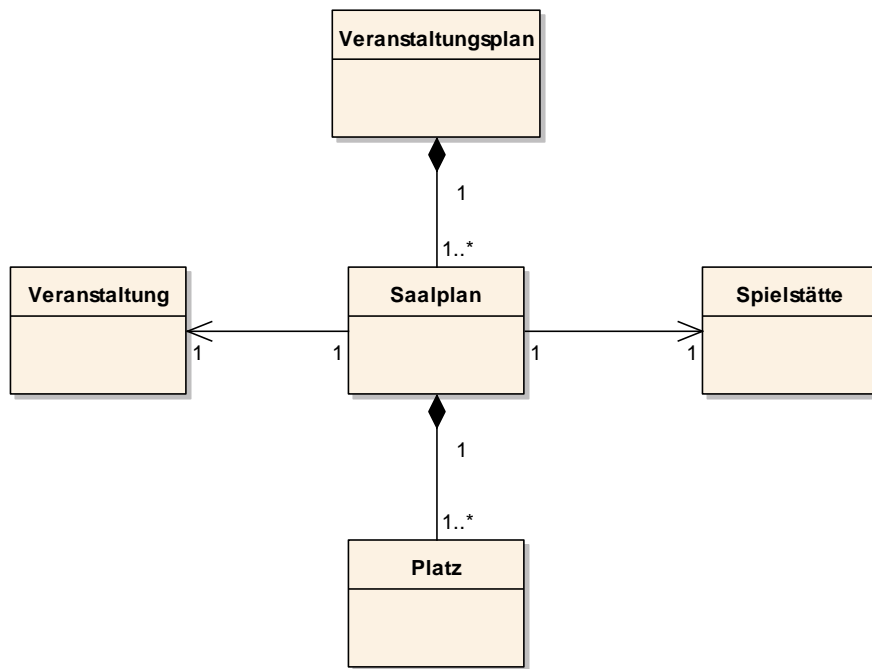


Abb. 51: Beispiel Klassendiagramm

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Eine Klasse wird in der UML dargestellt durch

- [a] ein Rechteck mit drei Bereichen: Klassenname, Methoden, Zustände.
- [b] ein dreifärbiges Rechteck mit Klassenname, Parametern, Methoden.
- [c] ein Rechteck mit drei Bereichen: Klassenname, Attribute, Methoden.

2. Das einem Attribut vorangestellte Symbol

- [a] „#“ kennzeichnet Attribute, die als Properties nach außen sichtbar sind.
- [b] „+“ bedeutet generelle Sichtbarkeit (auch von außerhalb der Klasse).
- [c] „-“ kennzeichnet Attribute mit einfachen Datentypen.

3. Ein Domainmodell ist

- [a] die Darstellung der Wirklichkeit in der Analysephase.
- [b] ein Modell basierend auf Attributen und Methoden von Klassen.
- [c] eine Basis für die Erzeugung von Source Code.

4. In einer Komposition

- [a] sind die Klassen als lose Teile miteinander verbunden.
- [b] ist eine Klasse ein untrennbarer Bestandteil der anderen.
- [c] erbt eine Klasse Attribute und Methoden von der anderen.

5. Bei der Generalisierung

- [a] zeigt der Pfeil der Verbindung auf die spezielle Klasse.
- [b] erbt die spezielle Klasse Methoden und Attribute der allgemeinen Klasse.
- [c] werden gleichwertige Attribute zwischen Klassen übergeben und gefüllt.

Richtige Antworten: 1c, 2b, 3a, 4b, 5b

Paketdiagramm (Package Diagram)

Ein Paket (Package) ist eine logische Ansammlung von Modellelementen beliebigen Typs, mit denen das Gesamtmodell in kleinere überschaubare Einheiten gegliedert wird.

Ein Paket definiert einen Namensraum, d. h., innerhalb eines Paketes müssen die Namen der enthaltenen Elemente eindeutig sein. Jedes Modellelement kann in anderen Paketen referenziert werden, gehört aber nur zu höchstens einem (Heimat-)Paket. Pakete können verschiedene Modellelemente enthalten, beispielsweise Klassen und Anwendungsfälle. Sie können hierarchisch gegliedert werden, d. h. ihrerseits wieder Pakete enthalten.

Ein Modellelement, beispielsweise eine Klasse, kann in verschiedenen Paketen benutzt werden, jedoch hat jede Klasse ihr Stammpaket. In allen anderen Paketen wird sie lediglich über ihren qualifizierten Namen

Paketname::Klassenname

zitiert. Dadurch entstehen Abhängigkeiten zwischen den Paketen, d. h., ein Paket nutzt Klassen eines anderen Pakets.

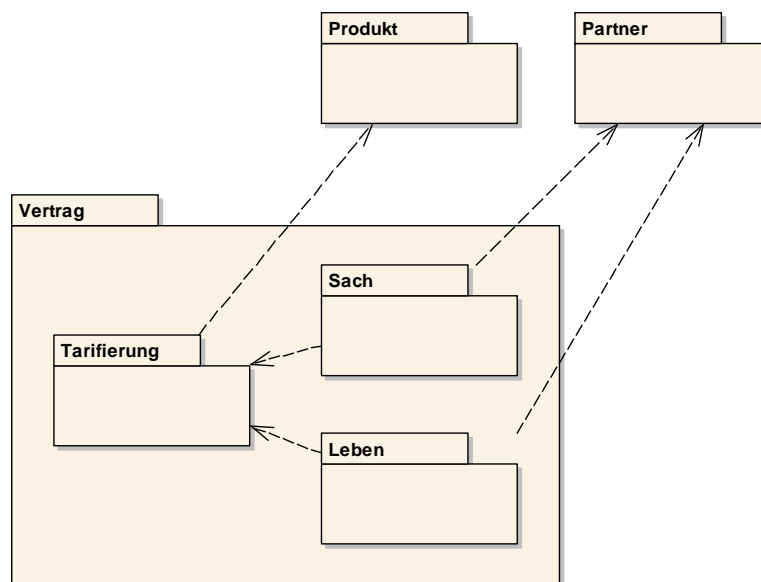


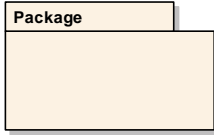



Abb. 52: Beispiel Paket Diagramm

Ein Paket wird in Form eines Aktenregisters dargestellt. Innerhalb dieses Symbols steht der Name des Paketes. Werden innerhalb des Symbols Modellelemente angezeigt, steht der Name auf der Registerlasche, andernfalls innerhalb des großen Rechtecks. Oberhalb des Paketnamens können Stereotype notiert werden.

Möchte man eine Arbeitsteilung auf mehrere Gruppen von Entwicklern vornehmen, kann man die Unterteilung in Pakete nutzen und jeder Gruppe ein solches Paket zur Bearbeitung übergeben. Damit die spätere Zusammenführung der Pakete problemlos möglich ist, wird eine Schnittstelle vereinbart. Dabei ist am Anfang nur der Name für die Schnittstelle nötig. Über die Schnittstelle können die im Paket enthaltenen Funktionen aufgerufen werden. Somit kann das Paket entsprechend getestet und die spätere Zusammenarbeit mit weiteren Paketen garantiert werden. Garantiert bedeutet in diesem Zusammenhang, dass für die in dem Paket implementierten Funktionen beim Aufruf mit möglichen Eingabewerten genau die definierten Ergebniswerte geliefert werden.

Das Zusammenspiel der Pakete des Systems wird in einem Paketdiagramm dargestellt, in dem man auch den internen Aufbau eines Pakets modelliert.

Grafische Elemente

Name/Element	Verwendung
Paket 	Das Symbol des Paketes ist einer Karteikarte nachempfunden. Innerhalb dieses Symbols wird der Paketname dargestellt. Möchten Sie innerhalb des Paketes die Kommunikationen der Unterpakete modellieren, können Sie den Paketnamen in das oben angefügte Rechteck verlagern.
Kommunikation 	Die Kommunikation der Pakete untereinander wird mit einem Pfeil dargestellt, der auf einer unterbrochenen Linie verläuft. Der Pfeil geht von dem Paket aus, von dem auch die Kommunikation überwiegend ausgeht.
Generalisierung 	Erbt ein Paket von einem anderem, wird das Symbol für die Generalisierung verwendet. Mit einem Pfeil werden die beteiligten Pakete verbunden, wobei die Pfeilspitze auf das Paket zeigt, von dem geerbt wird.
Enthält (Nesting) 	Mit diesem Symbol können Sie die Paketgliederung vom übergeordneten Paket her modellieren. Dazu wird das Symbol „Enthält“ an das Paketsymbol des übergeordneten Paketes gezeichnet. Die untergeordneten Pakete werden durch durchgezogene Linien mit diesem Symbol verbunden.

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. In einem Paket (Package)

- [a] werden ausschließlich Klassen und Objekte einer Komponente gesammelt.
- [b] dürfen keine weiteren Unterpakete vorkommen.
- [c] können beliebige Elemente zusammengefasst werden.

2. Ein Namespace

- [a] ist der Bereich, in dem alle Elemente einen eindeutigen Namen haben müssen.
- [b] wird durch senkrecht verlaufende Bahnen notiert.
- [c] wird durch einen Doppelpunkt getrennt hinter dem Objektnamen notiert.

Richtige Antworten: 1c, 2a

Interaktionsdiagramm (Interaction Diagram)

Eine Interaktion beschreibt eine Reihe von Nachrichten, die eine ausgewählte Menge von Beteiligten in einer zeitlich begrenzten Situation austauscht.

Zur Darstellung von Interaktionen definiert die UML 2.1 drei unterschiedliche Diagramme:

- Sequenzdiagramme (Sequence Diagram) betonen die Reihenfolge des Nachrichtenaustauschs.
- Kommunikationsdiagramme (Communication Diagram) betonen die Beziehungen zwischen den Beteiligten.
- Zeitdiagramme (Timing Diagram) betonen die Zustandswechsel der Beteiligten in Abhängigkeit von der Zeit und den ausgetauschten Nachrichten.

Sequenzdiagramm (Sequence Diagram)

Beim Sequenzdiagramm steht der zeitliche Verlauf der Nachrichten im Vordergrund. Die Reihenfolge der Nachrichten entspricht ihrer horizontalen Position im Diagramm. Es wird festgelegt, wann ein Objekt erstellt wird und wann Nachrichten zu welchem Objekt gesendet werden.

Die beteiligten Objekte werden durch ein Rechteck und eine senkrechte gestrichelte Linie repräsentiert. Beides zusammen wird Lebenslinie (Lifeline) genannt. Nachrichten (Messages) werden durch Pfeile zwischen den Lebenslinien dargestellt. Die Zeit verläuft von oben nach unten. Der zeitliche Verlauf der Nachrichten wird dadurch hervorgehoben.

Das Sequenzdiagramm in der folgenden Abbildung zeigt eine Interaktion von drei Objekten. Zu beachten ist, dass das ganze Diagramm eine Interaktion repräsentiert und eine Interaktion nicht nur ein einzelner Nachrichtenaustausch ist.

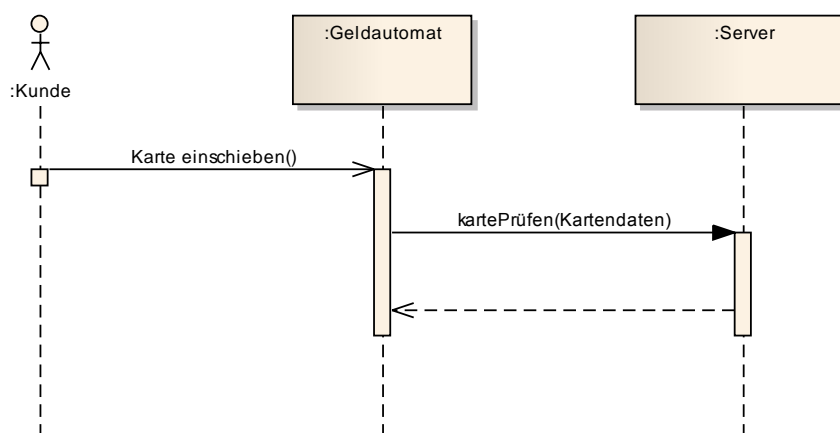


Abb. 53: Einfaches Beispiel eines Sequenzdiagramms

Im Kopf der Lebenslinie steht der (optionale) Elementname mit der zugehörigen Klasse in der üblichen Deklarationsnotation: name : typ.

Ausführungsfokus

Wenn zwischen Lebenslinien Nachrichten ausgetauscht werden, muss auch ein Verhalten in den zugehörigen Elementen ausgeführt werden. Das wird durch die länglichen Rechtecke auf der Lebenslinie dargestellt. Die Rechtecke repräsentieren den sogenannten Ausführungsfokus (ExecutionOccurence). Anfang und Ende des Ausführungsfokus sind durch sogenannte Ereignisauftritte definiert. Einfacher gesagt, das Senden und Empfangen von Nachrichten bestimmt Anfang und Ende des Ausführungsfokus.

Nachrichtenarten

Die Übertragung einer Nachricht wird mit Pfeilen notiert. Die Beschriftung der Nachrichten erfolgt mit den Namen der zugehörigen Operationen. UML kennt verschiedene Arten von Nachrichten-

ten, die durch unterschiedliche Pfeilnotationen dargestellt werden. In der nachfolgenden Abbildung werden die verschiedenen Nachrichtenarten und zugehörige Notationsformen gezeigt.

- *Synchrone Nachrichten* (Synchronous Messages) haben eine gefüllte Pfeilspitze. Synchron bedeutet, dass der Aufrufer wartet, bis das aufgerufene Verhalten beendet wurde. Die Antwort (Reply Message) auf einen synchronen Aufruf wird mit einer gestrichelten Linie und offener Pfeilspitze dargestellt.
- *Asynchrone Nachrichten* (Asynchronous Messages) haben eine offene Pfeilspitze. Asynchron bedeutet, dass der Aufrufer nicht wartet, sondern unmittelbar nach dem Aufruf fortfährt. Entsprechend gibt es auf asynchrone Aufrufe auch keine Antwortpfeile.
- *Verlorene Nachrichten* (Lost Messages) haben eine offene Pfeilspitze, die auf einen ausgefüllten Kreis zeigt. Der Kreis ist nicht mit einer Lebenslinie verbunden. Bei einer verlorenen Nachricht ist der Sender der Nachricht bekannt, aber nicht der Empfänger.
- *Gefundene Nachrichten* (Found Messages) haben eine offene Pfeilspitze. Die Linie geht von einem ausgefüllten Kreis aus. Der Kreis ist nicht mit einer Lebenslinie verbunden. Bei einer gefundenen Nachricht ist der Empfänger der Nachricht bekannt, aber nicht der Sender.
- Eine Nachricht, die ein neues Exemplar erzeugt, wird mit einer Linie und offener Pfeilspitze dargestellt. Die zu dem Exemplar gehörende Lebenslinie beginnt erst an dieser Stelle im Diagramm, d. h., der Pfeil zeigt auf den Kopf der Lebenslinie.

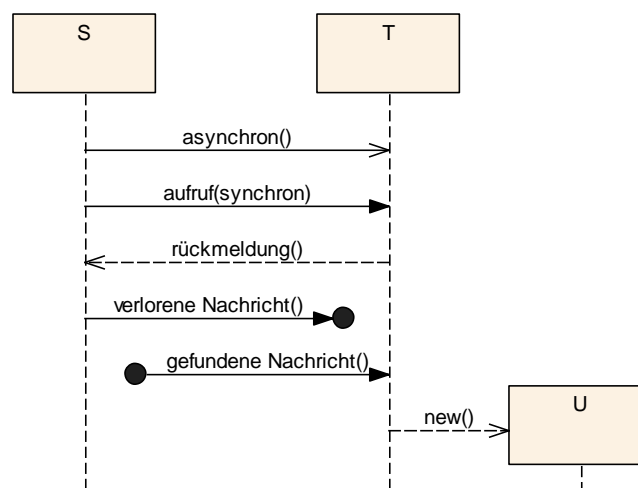


Abb. 54: Notationsformen der verschiedenen Nachrichtenarten

Durch Voranstellen des Zeichens * modelliert man das wiederholte Senden der Nachricht. Die Nachricht erhält in diesem Fall das Zeichen * vorangestellt.

Auf den Nachrichtenpfeilen wird die Nachricht notiert. Die Syntax lautet
[attribut =] name [(argumente)] [: rückgabewert]


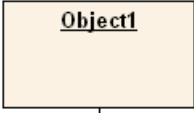
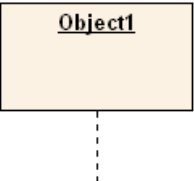

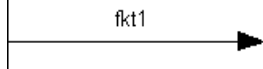
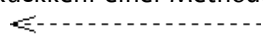
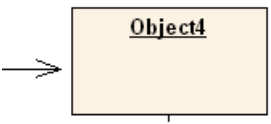

Wobei

- attribut eine lokale Variable der Interaktion oder ein Exemplar einer Lebenslinie sein kann. Die Attributzuweisung wird nur bei synchronen Nachrichten mit Rückgabewert verwendet.
- name der Name der aufzurufenden Nachricht oder der Name des Signals, das gesendet wird, ist. Das Senden eines Signals hat immer asynchronen Charakter.
- argumente eine kommaseparierte Liste der Parameterwerte ist, die der Nachricht übergeben werden.

Wird über das Absetzen einer Botschaft ein Objekt erzeugt (z. B. über den Aufruf der Methode `new`), beginnt die Lebenslinie des Objekts erst an dieser Position. Die Auflösung eines Objektes wird durch ein Kreuz auf der Lebenslinie dargestellt.

Symbole

Die folgende Tabelle enthält die Symbole der Sequenzdiagramme.

Symbol/Name	Verwendung
Systemgrenze 	Die Systemgrenze isoliert den betrachteten Programmteil vom übrigen Programm. Sie dient meist als Ausgangspunkt des auslösenden Methodenaufrufs. Nicht immer wird der Programmfluss von einem Objekt außerhalb des betrachteten Bereichs ausgelöst, sodass in diesem Fall keine Systemgrenze eingezeichnet werden muss.
Objekt 	Ein Objekt wird mit einem Rechteck, das den Namen enthält, dargestellt. Das Unterstreichen des Namens kann hier entfallen, da keine Verwechslung mit dem Klassennamen auftreten kann. Klassen werden in diesem Diagramm nicht dargestellt. Die Objekte werden entlang der oberen Blattkante nebeneinander angezeigt.
Lebenslinie 	Jedes Objekt liegt auf einer senkrechten Linie, der Lebenslinie. Die Lebensdauer des Objekts nimmt in Richtung der unteren Blattkante zu. Für Objekte, die zu Beginn des Programmteils bereits existieren, werden die Objektsymbole an der oberen Blattkante gezeichnet. Für Objekte, die innerhalb des Programmteils neu erstellt werden, wird das Symbol in Höhe des Methodenaufrufs gezeichnet, in dessen Folge das Objekt erstellt wird.
Objekt-Aktivität 	Ist ein Objekt an einem Methodenaufruf beteiligt, wird es aktiv. Die Lebenslinie verdickt sich. Ruft ein Objekt eine eigene Methode auf, verdickt sich die Lebenslinie ein weiteres Mal. Diese Aktivitäten werden nicht immer mitgezeichnet.
Methodenaufrufe 	Ruft ein Objekt eine Methode eines anderen Objekts auf, wird das über einen durchgehenden Pfeil symbolisiert, der auf das Objekt zeigt, dessen Methode aufgerufen wird. An dieses Symbol wird der Methodenname geschrieben. Diesem Namen kann in Klammern die Parameterliste angefügt werden.
Rückkehr einer Methode 	Prinzipiell werden nur die Methodenaufrufe in das Sequenzdiagramm eingezeichnet. Möchten Sie dennoch die Rückkehr der Methoden einzeichnen, können Sie dies mit einem Pfeil und unterbrochener Linie vornehmen.
Objekt-Erstellung 	Erstellt eine Methode ein Objekt, endet der Pfeil der Methode an dem rechteckigen Symbol des Objekts. Die Lebenslinie beginnt bei diesem Symbol.
Objekt-Zerstörung 	Wird durch den Aufruf einer Methode ein Objekt zerstört, endet die Lebenslinie des Objekts mit einem Kreuz unterhalb des Symbols des Methodenaufrufs.

Beispiel

In einem Ticketsystem für ein Schauspielhaus werden auf einer Internetseite Eintrittskarten aus dem Saalplan heraus verkauft. Der Saalplan verwaltet die Sitzplätze einer Veranstaltung.

Wird ein Platz von einem Besucher ausgewählt, ruft das betreffende Objekt `aPlatz` die Methode „kaufen“ des Objekts `aBestellung` auf und übergibt eine Referenz auf sich selbst im Parameter.

Das Objekt der Klasse *Bestellung* ruft die Methode „*istFrei*“ der Klasse *Saalplan* auf, um zu überprüfen, ob der im Parameter übergebene Platz noch frei ist. Ist der Platz noch frei, ruft das Saalplan-Objekt seine eigene Methode „*reservieren*“ auf. Damit wird der Platz zunächst reserviert.

Nachdem dies erfolgt ist, wird die Rechnung für den Platz erstellt. Dazu ruft das Saalplan-Objekt die Methode „*buchen*“ mit dem ausgewählten Platz als Parameter auf. Die Methode „*buchen*“ gehört zum Objekt *aVerkauf*. Dieses stellt eine Liste der Rechnungspositionen dar, was hier nicht modelliert ist. Nachdem die Rechnungspositionen vom Verkaufs-Objekt zusammengestellt wurden, ruft dieses die Methode „*erstelleRechnung*“ des Objekts *aBestellung* auf. Um dem Internetbesucher den Erfolg seines Kaufes mitzuteilen, ruft das Bestellungs-Objekt die Methode „*bestätigt*“ der Klasse *Platz* auf. Der Besucher bestätigt die Bestellung, und die Methode „*bestätigt*“ kehrt mit dem Wert „*true*“ (wahr) zurück. Die Bestellung ist abgeschlossen.

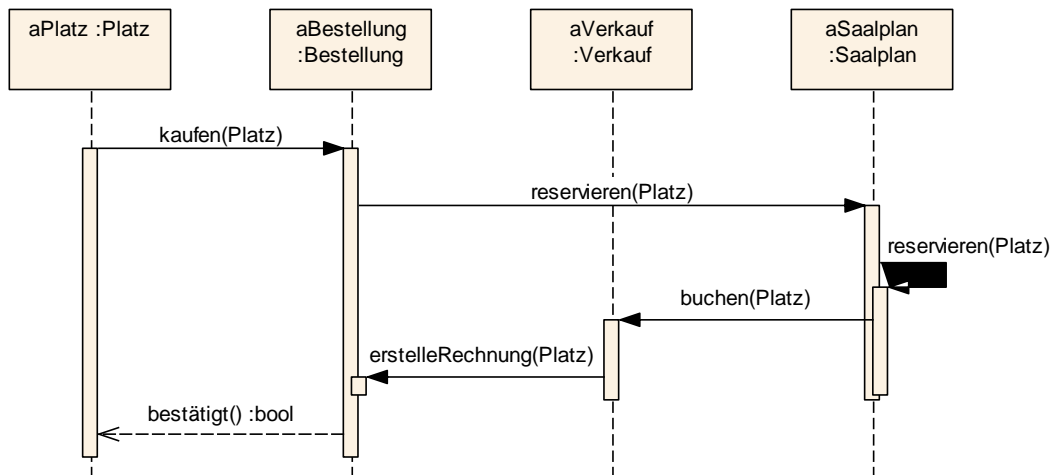


Abb. 55: Beispiel eines Sequenzdiagramms

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Interaktionsdiagramme beschreiben

- [a] Nachrichten, die Objekte innerhalb eines betrachteten Zeitrahmens austauschen.
- [b] die Kommunikation zwischen Attributen von Objekten.
- [c] Schnittstellen zum Benutzer und zu anderen Systemen.

2. Der Ausführungsfokus wird dargestellt durch

- [a] eine gestrichelte Linie unterhalb eines Objekts.
- [b] eine Verdickung der Lebenslinie eines Objekts.
- [c] einen Pfeil mit offener Spitze, der zwei Lebenslinien von Objekten verbindet.

3. Eine Nachricht, bei der der Aufrufer nicht auf einen Rückgabewert wartet,

- [a] heißt „asynchrone“ Nachricht.
- [b] wird im UML-Diagramm nicht modelliert.
- [c] wird durch ein „X“ auf der Lebenslinie des Objekts markiert.

4. Neue Objekte im Sequence Diagram werden

- [a] nur am oberen Blattrand modelliert, sie leben während der gesamten Interaktion
- [b] an der Position des ersten Methodenaufrufs modelliert.
- [c] mit Hilfe der „lost message“ modelliert.

Richtige Antworten: 1a, 2b, 3a, 4b

Kommunikationsdiagramm (Communication Diagram)

Das Kommunikationsdiagramm entspricht dem Kollaborationsdiagramm der UML 1.x. Es ist umbenannt worden, da der Name Kollaborationsdiagramm irreführend ist. Es gibt in der UML auch das Modellelement Kollaboration, das aber nichts mit Kollaborationsdiagrammen zu tun hat.

Das Kommunikationsdiagramm stellt die Fakten eines Sequenzdiagramms unter einer anderen Betrachtungsweise dar. In diesem Diagramm wird besonderes Augenmerk auf die Zusammenarbeit der Objekte untereinandergelegt. Dazu werden ausgewählte Nachrichten verwendet, mit denen die zeitliche Abfolge der Kommunikation zwischen den Objekten erfolgt. Es stellt somit in kompakterer Form die Aussagen des Sequenzdiagramms zusammen.

In der folgenden Abbildung ist gut zu sehen, dass das Kommunikationsdiagramm die Beziehungen zwischen den Beteiligten in den Vordergrund stellt und nicht wie das Sequenzdiagramm die zeitliche Abfolge des Nachrichtenaustauschs.

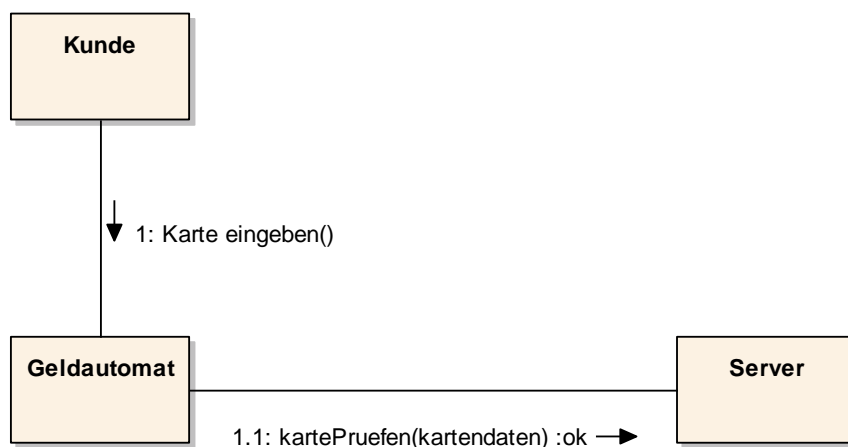


Abb. 56: Beispiel Kommunikationsdiagramm „Verfügungsberechtigten identifizieren“

Die grafische Darstellung besteht aus einem Rechteck, das den Objektnamen und die zugehörige Klasse enthält. Ein Doppelpunkt trennt beide Namen. Die Objekte werden mit Assoziationslinien verbunden. Ein kleiner Pfeil zeigt jeweils die Richtung der Nachricht vom Sender zum Empfänger an. Wenn mit der Nachricht Argumente übergeben werden, sind diese aufgeführt. Mögliche Rückgabewerte können ebenfalls in der Form

antwort := Nachrichtenname (Parameterliste)

angegeben werden.

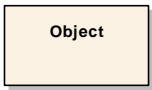

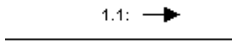
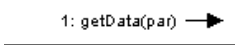
Um den zeitlichen Ablauf zu modellieren, erhalten die Nachrichten Nummern. Durch eine Nachricht können weitere Nachrichten ausgelöst werden. Sie erhalten Unternummern der auslösenden Nachricht (z. B. 1.2).

Wird eine Nachricht mehrfach ausgelöst, kann durch ein Zeichen * vor dem Nachrichtennamen diese Iteration modelliert werden.

Objekte, die innerhalb des dargestellten Szenarios erzeugt werden, können Sie mit dem Stereotyp *new* kennzeichnen. Für Objekte, die innerhalb des dargestellten Szenarios zerstört werden, erfolgt die Bezeichnung mit dem Stereotyp *destroy*. Objekte, die innerhalb des Szenarios erzeugt und zerstört werden, erhalten das Stereotyp *transient*.

Symbole

Die folgende Tabelle enthält die Symbole der Kommunikationsdiagramme.

Symbol/Name	Verwendung
Objekt 	Das Rechteck ist das Symbol eines Objekts und enthält den Objektnamen. Da keine Verwechslung mit Klassen eintreten kann, muss der Name nicht unterstrichen sein.
Kommunikation 	Kommunizieren zwei Objekte über einen Methodenaufruf miteinander, wird diese Verbindung mit einer durchgängigen Linie dargestellt, die beide Objekte verbindet.
Kommunikationsrichtung 	Zusätzlich zur Verbindungslinie wird nach dem Methodennamen durch einen Pfeil angezeigt, zu welchem Objekt die Methode gehört. Die Pfeilspitze zeigt auf dieses Objekt.
Beschriftung der Kommunikationslinien 	Die Linie wird mit dem Methodennamen und der Parameterliste beschriftet. Eine Nummerierung wird zur Kennzeichnung der zeitlichen Reihenfolge der Methodenaufrufe vor dem Methodennamen angezeigt und durch einen Doppelpunkt von diesem getrennt. In einer Bedingung können Sie die Nummern der Methoden angeben, die als Voraussetzung bereits abgearbeitet wurden. Das Zeichen * vor dem Methodennamen kennzeichnet eine Wiederholung der Methode. Die Bedingung für die Wiederholung können Sie nach dem Zeichen * angeben.

Beispiel

Das Beispiel modelliert den Kauf eines Tickets über das Internet. Der Internetkunde wählt einen Platz auf der Internetseite eines Schauspielhauses aus und ruft damit die Methode *auswählen* auf. Die Interaktion wird gestartet. Das Objekt *Platz* ruft die Methode *kaufen* des *Bestellung*-Objekts auf und übergibt eine Referenz auf sich selbst im Parameter. Das Objekt *Bestellung* ruft die Methode *isFrei* des *Saalplan*-Objekts auf. Diese Methode ruft zum einen die Methode *reservieren* und zum anderen die Methode *buchen* auf. Sie unterscheiden sich deshalb nur in der Unter-Nummer. Das *Saalplan*-Objekt ruft die Methode *buchen* des Objekts *Verkauf* auf. Dieses Objekt ruft die Methode *erstelleRechnung* des *Bestellung*-Objekts auf und übergibt die Rechnungsposition des gebuchten Platzes. Nachdem die Rechnung vom Objekt *Bestellung* erstellt wurde, erfolgt die Benachrichtigung des Internetkunden über die erfolgreiche Buchung. Dazu wird die Methode *bestätigt* aufgerufen, die die Bestätigung des Kunden erfragt und zurückgibt.

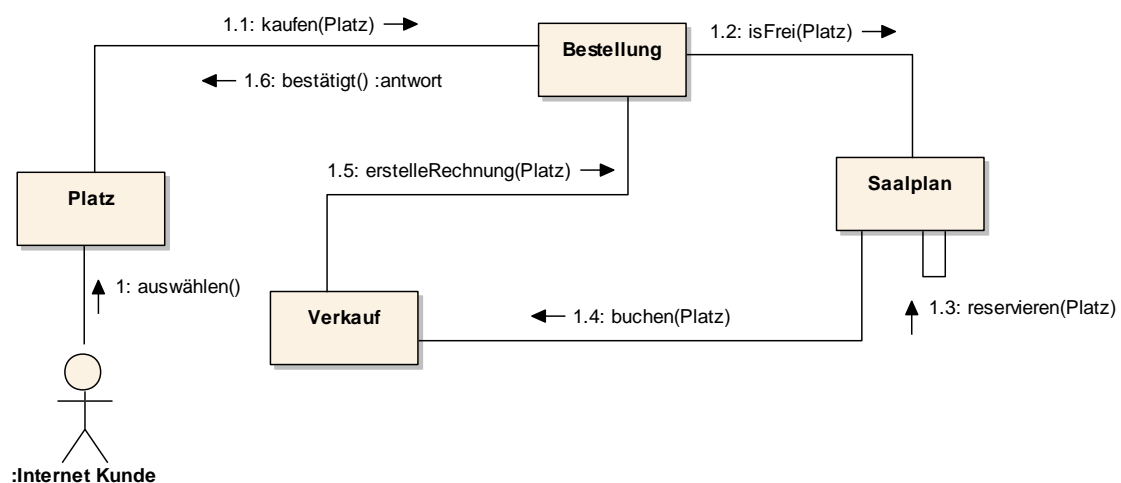


Abb. 57: Beispiel Kommunikationsdiagramm „Ticketkauf über Internet“

Sequenzdiagramme vs. Kommunikationsdiagramme

Sequenz- und Kommunikationsdiagramme sind sehr ähnlich und können auch in einigen UML-Tools ineinander übergeführt werden. Der Fokus beim Sequenzdiagramm liegt auf dem zeitlichen Aspekt, beim Kommunikationsdiagramm liegt er hingegen auf den Beziehungen zwischen den Objekten. Der größte Vorteil der Sequenzdiagramme und gleichzeitig größte Nachteil der Kommunikationsdiagramme ist der deutlich sichtbare zeitliche Ablauf, der bei Kommunikationsdiagrammen zwar prinzipiell auch durch ein Nummerierungsschema visualisierbar ist, aber weniger deutlich sichtbar. Dafür muss beim Erstellen nicht sofort die Ausführungsreihenfolge festgelegt werden, was die Erstellung der Sequenzdiagramme manchmal etwas trickreich gestaltet. Andererseits ist in einem umfangreichen Kommunikationsdiagramm die Reihenfolge schlecht lesbar, der Leser muss durch eine *Suche* herausfinden, ob z. B. auf 1.1 2, 1.2 oder gar 1.1.1 folgt!

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Sequenz- und Kommunikationsdiagramm

- [a] haben nichts miteinander zu tun.
- [b] tauschen Nachrichten über „lost“ und „found messages“ untereinander aus.
- [c] sind einander ähnlich und können mit diversen Tools ineinander übergeführt werden.

2. Der zeitliche Ablauf ist im Kommunikationsdiagramm

- [a] über ein Nummerierungsschema sichtbar.
- [b] durch die Namen der Methodenaufrufe sichtbar.
- [c] besser sichtbar als im Sequenzdiagramm.

3. Die Kommunikationsrichtung zwischen Objekten im Kommunikationsdiagramm

- [a] wird mit einem offenen Pfeil an der Assoziation modelliert.
- [b] wird durch gestrichelte und durchgezogene Nachrichten gekennzeichnet.
- [c] wird durch einen geschlossenen Pfeil neben dem Methodennamen symbolisiert.

4. In den Interaktionsdiagrammen werden Objektnamen

- [a] durch die Verwechslungsgefahr mit Klassen immer unterstrichen dargestellt.
- [b] oft ohne Unterstreichung gezeigt, es besteht keine Verwechslungsgefahr mit Klassen.
- [c] ohne Unterstreichung modelliert, dafür immer mit führendem Paket- und Klassennamen.

Richtige Antworten: 1c, 2a, 3c, 4b

Interaktionsübersichtsdiagramm (Interaction Overview Diagram)

Das Interaction Overview Diagram ist neu in UML 2.0/2.1. Es stellt jedoch lediglich eine Mischung aus Aktivitäts- und Sequenzdiagramm dar, wobei sowohl Aktivitätsblöcke in ein Sequenzdiagramm gemischt werden können, als auch umgekehrt. Da es darüber hinaus keinerlei Neuheiten enthält, wird hier nicht näher darauf eingegangen.

Das folgende Beispiel zeigt den Ablauf „Überweisung tätigen“, wobei hier das Hauptdiagramm ein Aktivitätsdiagramm ist und eine Aktion (Auftrag speichern) durch ein Sequenzdiagramm genauer beschrieben wird.

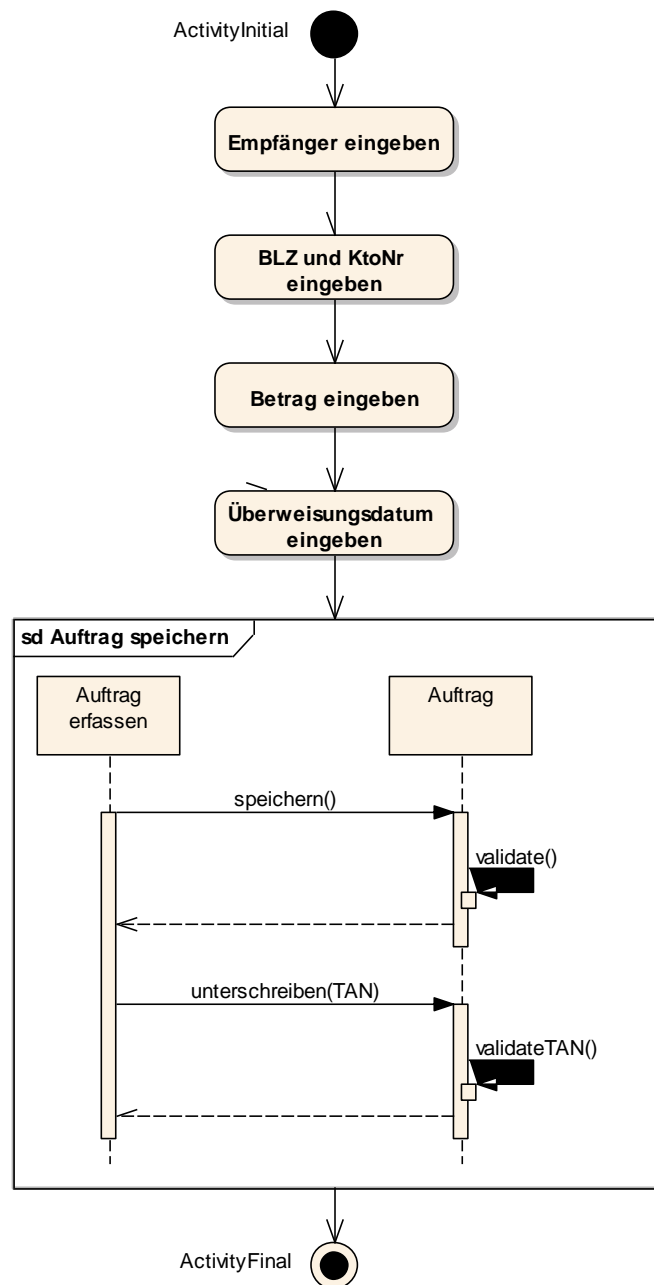


Abb. 58: Beispiel Interaction Overview Diagram

Komponentendiagramm (Component Diagram)

Komponentendiagramme zeigen die Beziehungen der Komponenten untereinander. Eine Komponente ist eine ausführbare und austauschbare Softwareeinheit mit definierten Schnittstellen und eigener Identität. Eine Komponente ist ähnlich einer Klasse instanzierbar und kapselt eine komplexe Funktionalität.

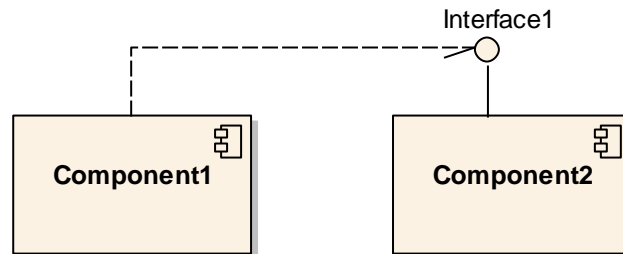


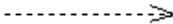


Abb. 59: Notation Komponentendiagramm

Die Komponente *Component2* stellt über ihre Schnittstelle eine Funktionalität zur Verfügung. Die Komponente *Component1* kommuniziert über die von der Komponente *Component2* angebotene Schnittstelle *Interface1* und verwendet deren Funktionalität.

Symbole

Die folgende Tabelle enthält die Symbole der Komponentendiagramme.

Name/Symbol	Verwendung
Komponente 	Das Symbol der Komponente besteht aus einer Anordnung von drei Rechtecken. Zwei Rechtecke werden auf der linken Seite des dritten, größeren Rechtecks gezeichnet. In diesem steht der Name der Komponente. Die Komponenten erfüllen vollständig die Aufgaben eines abgesteckten Bereiches, für die sie entwickelt wurden.
Schnittstelle 	<p>Eine Komponente kann eine Schnittstelle anbieten. Über diese erfolgt ein verwalteter Zugriff auf die Funktionalität der Komponente. Die Schnittstellen machen diese Elemente austauschbar.</p> <p>Eine Schnittstelle wird mit einem Kreis dargestellt. Dieser Kreis wird mit der Klasse, dem Paket oder der Komponente durch eine durchgezogene Linie verbunden. Der Name der Schnittstelle wird neben dem Symbol angegeben.</p>
Abhängigkeit 	Der Pfeil mit gestrichelter Linie symbolisiert den Zugriff auf die Schnittstelle. Der Pfeil zeigt auf den Kreis des Schnittstellensymbols. Er kann aber auch direkt auf eine Komponente zeigen, wenn nicht über die Schnittstelle zugegriffen wird.

Beispiel

Im Beispiel werden vier Komponenten zu einer Anwendung verknüpft, um Daten einer Datenbank in einem Fenster anzuzeigen. Die Datenbank-Verbindungs-Komponente ist für den Aufbau, die Verwaltung und den Abbau der Verbindung zur Datenbank verantwortlich. Über eine Schnittstelle stellt sie Methoden zur Verfügung, über welche die Daten-Container-Komponente Daten aus der Datenbank anfordern kann.

Die Daten-Container-Komponente verwaltet die angeforderten Datensätze. Gleichzeitig wird bei jedem Datentransfer eine Transaktion über die Schnittstelle der Transfersteuerungskomponente gestartet. Die Komponente für die grafische Darstellung ruft über die Schnittstelle der Daten-Container-Komponente die Datensätze ab und stellt sie auf dem Bildschirm dar.

Ändert der Anwender die Daten, so übergibt die Komponente die Änderungen an die Daten-Container-Komponente. Die geänderten Daten werden von dieser über die Verbindungs-komponente in die Datenbank geschrieben und die neuen Daten werden über die Transfersteuerungskomponente bestätigt, woraufhin die gestartete Transaktion beendet wird.

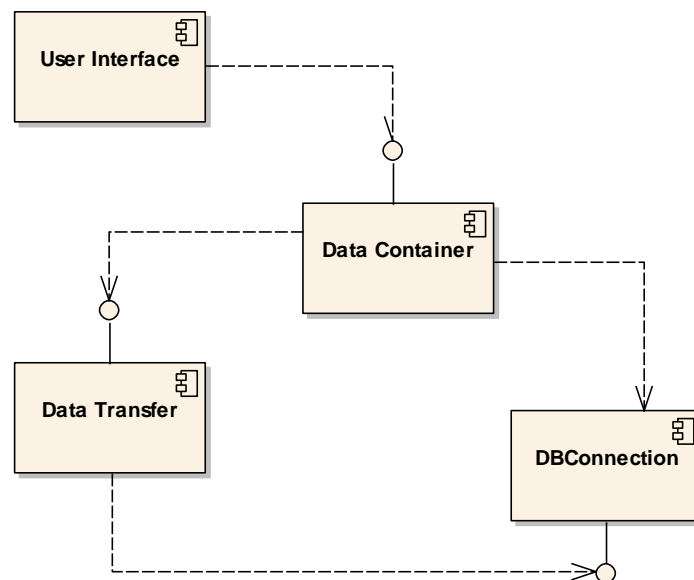


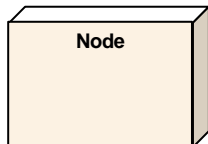
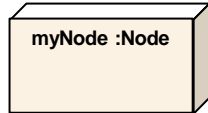



Abb. 60: Beispiel Komponentendiagramm

Verteilungsdiagramm (Deployment Diagram)

In diesem Diagramm werden Abhängigkeiten zwischen Knoten, die Knoten selbst, deren Inhalte und Kommunikationen gezeigt. Ein Knoten ist eine physische Einheit, die über Speicherplatz und Rechenleistung verfügt (z. B. ein PC) auf denen wiederum Komponenten laufen. Die Knoten werden als Quader gezeichnet. In den Quadern können Komponenten oder Laufzeitobjekte (Prozesse) eingetragen werden. Statt der Quader können auch Bilder verwendet werden, die den Knoten grafisch darstellen. Verteilungsdiagramme zeigen, welche Komponenten und Objekte auf welchen Knoten (Computer, Prozesse) laufen, d. h., wie diese konfiguriert sind und welche Kommunikationsbeziehungen bestehen.

Symbole

Die folgende Tabelle enthält die Symbole der Einsatz- und Verteilungsdiagramme.

Symbol/Name	Verwendung
Knoten 	Ein Knoten wird mit einem Quader dargestellt. Die Beschriftung, die innerhalb des Quaders angegeben wird, erhält den Namen des Knotens, z. B. einen Rechnernamen, einen Client- oder einen Prozessnamen.
Teilsystem 	Wird auf einem Knoten eine Anwendung im Speicher ausgeführt, wird diese Anwendung als Knoteninstanz modelliert. Zur Unterscheidung mit dem Knotensymbol wird die Beschriftung im Quader unterstrichen. Dieses Symbol wird in das Knotensymbol eingefügt, auf dem der Teilprozess abläuft.
Komponente 	Eine Komponente ist vom Umfang kleiner als ein Teilsystem, übernimmt aber für einen festgelegten Anwendungsbereich die Aufgaben der Anwendung. Das Komponentensymbol wird in das Knotensymbol eingefügt, auf dem die Komponente installiert wurde.
Komponenteninstanz 	Die Komponenteninstanz ist ein Sammelbegriff für die Objekte, deren Klassen zur Datenstruktur der Komponente gehören und von dieser instanziiert werden. Das Symbol unterscheidet sich nicht vom Symbol der Komponenten. Im Unterschied zum Komponentensymbol wird der Name der Instanz unterstrichen dargestellt. Das Symbol wird in das Komponentensymbol eingefügt, dessen Instanz es darstellt.
Beziehungslinie 	Die Beziehungen zwischen den Knoten werden mit einer durchgezogenen Linie dargestellt, die die Knoten verbindet.

Beispiel

Bei einem Ticketsystem für Lichtspieltheater kann aus verschiedenen Anwendungen heraus auf die Ticketdaten zugegriffen werden. Die Daten liegen auf einem zentralen Server, der mit weiteren PCs vernetzt und mit dem Internet verbunden ist. Auf dem Server ist eine Komponente zur Administration der Datenbank installiert. Der Datenbankserver Interbase ist auf dem Server eingerichtet.

- Der Verkauf an der Abendkasse wird über einen PC in der Kassenhalle abgewickelt. Damit dieser sehr schnell auf die Daten zugreifen kann, wurde eine Clientanwendung entwickelt, die speziell auf die Anforderungen des Verkaufs an der Abendkasse angepasst wurde.
- Der Systembetreuer hat für seine Aufgaben einen PC, z. B. im Bereich der Buchhaltung. Auch er hat eine Clientanwendung, die es ihm ermöglicht, Veranstaltungen anzulegen und auszuwerten.
- Die Vorverkaufsstelle wird mit einer eigenen Clientanwendung betrieben, die über das Internet mit dem Server verbunden ist.
- Der Internetkunde ruft von einem PC aus Daten vom Ticketsystem über das Internet ab. Dafür ist eine auf diese Kunden zugeschnittene Clientanwendung des Ticketsystems nötig.

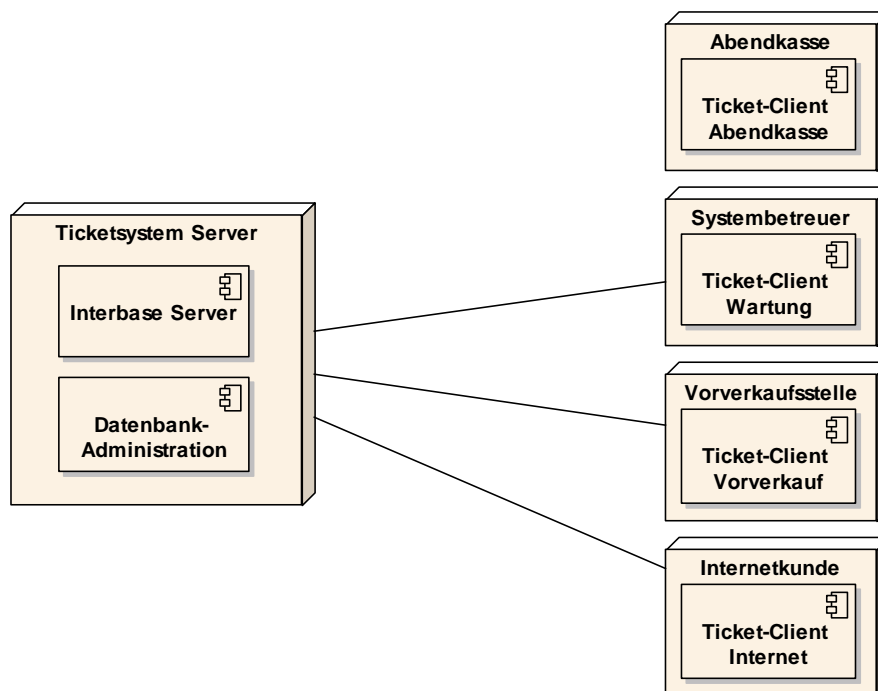


Abb. 61: Beispiel Verteilungsdiagramm

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Das Interaction-Overview-Diagramm

- [a] ist eine Mischung aus Communication- und Sequenzdiagramm.
- [b] stellt Knoten und auf ihnen befindliche Systemteile dar.
- [c] ist eine Mischung aus Activity- und Sequenzdiagramm.

2. Um gekapselte Softwareeinheiten mit definierter Funktionalität zu modellieren,

- [a] bedient man sich der instanziierten Klassen des Klassendiagramms.
- [b] verwendet man Komponenten und Interfaces im Komponentendiagramm.
- [c] setzt man am besten Packages und Assoziationen ein.

3. Eine von einer Komponente angebotenen Schnittstelle

- [a] erlaubt den Zugriff auf die Funktionalität der Komponente.
- [b] wird von einem großen Rechteck mit zwei kleinen Rechtecken symbolisiert.
- [c] hilft, die Komponente anhand ihres Namens im System eindeutig zu halten.

4. Ein Knoten im Deployment Diagram ist

- [a] eine physische Einheit, die über Speicherplatz und Rechenleistung verfügt.
- [b] ein Punkt, an dem Informationen zusammenlaufen und gespeichert werden.
- [c] eine Einheit, in die eine fertige Software unterteilt werden kann.

Richtige Antworten: 1c, 2b, 3a, 4a

Zeitdiagramm (Timing Diagram)

Das Timing Diagram wird vor allem in der hardwarenahen Programmierung oder in zeitlich kritischen Organisationsprojekten eingesetzt, um die Prozesse anhand der Zeitachse auf einen optimalen Ablauf hin zu untersuchen.

Die folgende Abbildung zeigt ein Zeitdiagramm. Hier steht der Zustand der Beteiligten in Bezug zu einer linearen Zeitachse im Vordergrund.

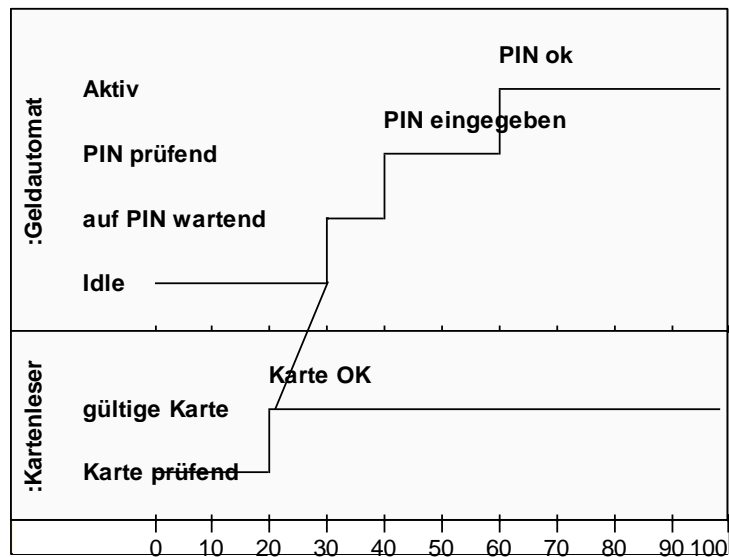


Abb. 62: Beispiel Zeitdiagramm

Anmerkung: Das Sequenzdiagramm besitzt zwar keine lineare Zeitskala ist aber auch durch die Verwendung von vertikalen Zeitmaßfeilen geeignet, Zeitabhängigkeiten exakt hervorzuheben.

Kompositionsstrukturdiagramm (Composite Structure Diagram)

Die interne Struktur einer Klasse kann mit dem Kompositionsstrukturdiagramm beschrieben werden. Das Diagramm ist neu in der UML 2.0 / 2.1. Auf den ersten Blick kann das Diagramm leicht mit einem Kommunikationsdiagramm verwechselt werden. Die folgende Abbildung zeigt ein Kompositionsstrukturdiagramm und ein zugehöriges Klassendiagramm mit Kompositionen. Die Aussage beider Diagramme ist gleichbedeutend.

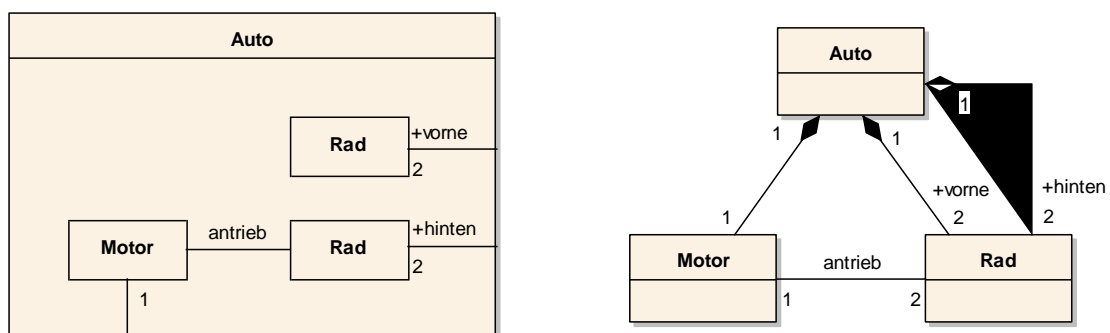


Abb. 63: Kompositionsstrukturdiagramm und äquivalentes Klassendiagramm

Objektdiagramm (Object Diagram)

Das Objektdiagramm hat eine gewisse Ähnlichkeit mit dem Klassendiagramm, mit dem entscheidenden Unterschied, dass hier nur Instanzen und keine Klassen dargestellt werden. Es zeigt einen bestimmten Ausschnitt des Programms zur Laufzeit. Es werden die einzelnen Objekte dargestellt, ebenso die Verbindungen und auch die Multiplizitäten. Eingesetzt wird dieser Diagrammtyp beispielsweise bei der Nachstellung eines Fehlers im laufenden System. Wenn nur unter bestimmten Voraussetzungen ein bestimmtes Verhalten der Software auftritt, kann das mit dem Objektdiagramm beschrieben werden, indem die für diesen Sachverhalt relevanten Eigenschaften und deren Werte in den Objekten dargestellt werden.

Im folgenden Beispiel ist links ein Klassendiagramm abgebildet und rechts das zugehörige Objektdiagramm.

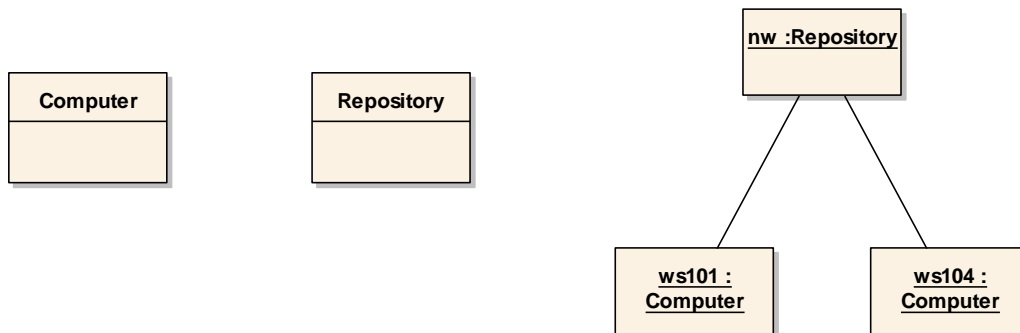


Abb. 64: Beispiel Objektdiagramm (rechts) und zugehöriges Klassendiagramm

Kapitelrückblick

Finden Sie die richtigen Antworten:

1. Das Timing Diagram wird nicht

- [a] in der hardwarenahen Programmierung eingesetzt.
- [b] für die Analyse zeitkritischer Organisationsprojekte verwendet.
- [c] als Darstellung der zeitlichen Abfolge von Aktivitäten eines Users herangezogen.

2. Die relevante Beziehung, die mittels Kompositionsstrukturdiagramm aufgelöst wird,

- [a] heißt Komposition und findet sich im Klassendiagramm.
- [b] heißt Kollaboration und findet sich im Komponentendiagramm.
- [c] heißt Komposition und findet sich im Strukturdiagramm.

3. Ein Objektdiagramm hilft

- [a] Fehler in einem System zu finden, die zur Laufzeit auftreten.
- [b] die geschickten Nachrichten zwischen Objekten darzustellen.
- [c] einen Überblick über die Systemlandschaft zu erhalten.

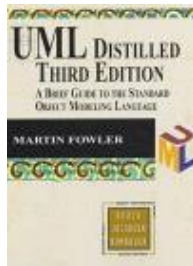
Richtige Antworten: 1c, 2a, 3a

Weiterführende Literatur - Empfehlungen



UML Kompakt
von Heide Baltzert

Zusammenfassung der Essenzen von UML auf 70 Seiten.
Kompaktes Nachschlagewerk für den Projektalltag



UML Distilled. Third Edition
by Martin Fowler

Der Klassiker unter den UML-Büchern.
Kurze, klare Präsentation der wichtigsten Features von UML



Car Multimedia Systeme Modell-Basiert Testen Mit Sysml
von Oliver Alt SysML



Objektorientierte Softwareentwicklung
von Bernd Oestereich

Die Einführung in objektorientierte Analyse und Design auf der Basis der
Unified Modeling Language



UML 2 in 5 Tagen
Von Heide Baltzert

Zusammenfassung der Essenzen von UML auf 70 Seiten. Kompaktes Nach-
schlagewerk für den Projektalltag



UML 2 Zertifizierung
 von *Tim Weilkiens, Bernd Oestereich*

Unterlagen zur UML Zertifizierungsprüfung der OMG (OCUP)



Real Time UML
 by *Bruce Powel Douglass*

Der Klassiker für Embedded- und Real Time Entwickler!



UML@Work - 3. Auflage
 von *Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger*

Sowohl für UML-Neulinge als auch für UML-Kenner. Sprachkonzepte und Notation der Diagrammarten zur Struktur- und Verhaltensmodellierung auf Basis eines durchgängigen Beispiels.



UML 2 glasklar
 von *Chris Rupp, Stefan Queins und Barbara Zengler*

Das Buch gibt einen umfassenden Überblick über alle 13 Diagrammtypen und stellt den nötigen Praxisbezug her, um UML 2 effektiv für eigene Projekte einzusetzen.



Systems Engineering mit SysML/UML
 vom *Tim Weilkiens*

Besonders interessant ist der Praxisbezug zur Analysephase, es werden auch Vorgehensweisen für Planungsmeetings vorgestellt....



UML2 projektorientiert
 Patrick Grässle / Henriette Baumann / Phlippe Bauman

Hier steht der Projektbezug im Vordergrund...



**Requirements-Engineering und -Management
von Chris Rupp & die SOPHISTen – 5. Auflage**

Das Standardwerk zum Thema Anforderungsanalyse. Es beschreibt den Prozess, Anforderungen an Systeme zu erheben und ihre ständige Veränderung zu managen.



**Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML
von Peter Hruschka und Chris Rupp**

Ein Klassiker, leider nicht mehr im Handel – aber als e-Book erhältlich bei <http://www.sophist.de/publikationen.html>



**Objektorientierte Softwareentwicklung - Analyse und Design mit der UML
von Bernd Österreich**



**Die UML-Kurzreferenz 2.3 für die Praxis: kurz, bündig, ballastfrei
von Bernd Österreich und Stefan Bremer**



**Objektorientierte Geschäftsprozessmodellierung mit der UML
von Bernd Österreich, Christian Weiss, Claudia Schröder, Tim Weilkiens und Alexander Lenhard**



**UML 2.0 Zertifizierung: Fundamental, Intermediate und Advanced
von Tim Weilkiens und Bernd Österreich**



Objektorientiertes Testen und Testautomatisierung in der Praxis.
Konzepte, Techniken und Verfahren
von *Uwe Vigerschow*



OEP - oose Engineering Process: Vorgehensleitfaden für agile
Softwareprojekte
von *Bernd Österreich, Claudia Schröder, Markus Klink und Guido Zockoll*



Soft Skills für Softwareentwickler: Fragetechniken, Konfliktmanagement,
Kommunikationstypen und -modelle
von *Uwe Vigerschow, Björn Schneider und Ines Meyrose*

Abbildungsverzeichnis

Abb. 1: Forward-, Reverse und Round-trip Engineering	5
Abb. 2: Historische Entwicklung der UML	7
Abb. 3: Beispiel Diagrammrahmen	8
Abb. 4: UML-Diagrammarten im Überblick	8
Abb. 5 Anwendungsfall	11
Abb. 6: Notation von Akteuren	12
Abb. 7: Notation von Anwendungsfällen	13
Abb. 8 System	13
Abb. 9 Multiplizität und Aktive/Passive Akteure	13
Abb. 10: Beispiel «include» Beziehung	14
Abb. 11: Beispiel «extend» Beziehung	15
Abb. 12: Beispiel «extend» Beziehung mit Extension Point & Condition	15
Abb. 13 Generalisierung von Use Cases	16
Abb. 14 Generalisierung von Akteuren	16
Abb. 15 Notizen	16
Abb. 16 Beispiel Use Case Diagramm	18
Abb. 17: Beispiel für eine Aktivität „Produktion von Sechserpacks“	20
Abb. 18 Kontrollfluss / Objektfluss	22
Abb. 19 Explizite vs. Implizite Decision	22
Abb. 20 Mergen	22
Abb. 21 Splitting und Synchronisation	23
Abb. 22 Join Specification	23
Abb. 23 Aufruf einer Aktivität mittels Aktion	24
Abb. 24 Strukturierte Aktivitäten	24
Abb. 25 Send/Receive	25
Abb. 26 Unterbrechungsbereich	25
Abb. 27: Beispiel: Prozess zur Durchführung einer Feier	29
Abb. 28: Beispiel State-Machine-Diagramm	33
Abb. 29: Beispiel Zustandsdiagramm „Bankomathochlauf“	35
Abb. 30: Beispiel Klasse	37
Abb. 31: Beispiel Stereotype	38
Abb. 32: Parametrisierbare Klasse	38
Abb. 33: Beispiel Objekte	39
Abb. 34: Assoziation und Komposition mit allen Angaben	40
Abb. 35: Assoziationen	40
Abb. 36: Multiplizität vs. Kardinalität	41
Abb. 37: Assoziationsklasse	41
Abb. 38: Assoziationsknoten (n-äre Assoziation)	42
Abb. 39: Notation Aggregation	42
Abb. 40: Beispiel Aggregation	42
Abb. 41: Aggregation und Komposition	43
Abb. 42: Beispiel Komposition	43
Abb. 43: Beispiel Aggregation und Komposition	43
Abb. 44: Beispiel Vererbung	44
Abb. 45: Beispiel Schnittstelle	46
Abb. 46: Notation bereitgestellte Schnittstelle	47
Abb. 47: Notation angeforderte Schnittstelle	47
Abb. 48: Notationsmöglichkeiten für Schnittstellen (Nutzung und Bereitstellung)	48
Abb. 49: Erweiterung von Schnittstellen	48
Abb. 50: Beispiel zur Implementierung von Schnittstellen	49
Abb. 51: Beispiel Klassendiagramm	51
Abb. 52: Beispiel Paket Diagramm	53
Abb. 53: Einfaches Beispiel eines Sequenzdiagramms	56
Abb. 54: Notationsformen der verschiedenen Nachrichtenarten	57
Abb. 55: Beispiel eines Sequenzdiagramms	59
Abb. 56: Beispiel Kommunikationsdiagramm „Verfügungsberechtigten identifizieren“	61
Abb. 57: Beispiel Kommunikationsdiagramm „Ticketkauf über Internet“	62
Abb. 58: Beispiel Interaction Overview Diagram	65
Abb. 59: Notation Komponentendiagramm	66
Abb. 60: Beispiel Komponentendiagramm	67

Abb. 61: Beispiel Verteilungsdiagramm	69
Abb. 62: Beispiel Zeitdiagramm	71
Abb. 63: Kompositionsstrukturdiagramm und äquivalentes Klassendiagramm	71
Abb. 64: Beispiel Objektdiagramm (rechts) und zugehöriges KlassendiagrammKapitelrückblick..	72

Index

- «call» 45
- «create» 45
- «derive» 45
- «instantiate» 45
- «permit» 45
- «realize» 45
- «refine» 45
- «trace» 45
- «use» 45
- Abhängigkeiten 44
- Abstrakte Klasse 38
- Activity Diagram 20
- Aggregation 42
- Akteur 11
- Aktivität 20
- Aktivitätsdiagramm 20
- Amigos 6
- Anwendungsfall 12
- Anwendungsfalldiagramm 11
- Assoziation 39
- Assoziationsklasse 41
- Asynchrone Nachrichten* 57
- Asynchrone Prozesse 25
- Attribute 39
- Ausführungsfokus 56
- Beziehungen zwischen Klassen 39
- Booch 6
- Call Behaviour Action* 23
- Class Diagram 37
- Collaboration Diagram 61
- Communication Diagram 61
- Component Diagram 66
- Composite Structure Diagram 71
- ControlFlow* 21
- David Harel 32
- Dependencies 44
- Deployment Diagram 68
- Diagrammeinsatz 9
- Diagrammtypen 8
- Eigenschaften 39
- Element verlinken 24
- Elemente strukturieren 24
- Enthält-Beziehung 14
- Erweiterungsbeziehung 14
- Extend 14
- Gefundene Nachrichten 57
- Generalisierung 16, 44
- Grundlagen der Verhaltensmodellierung 10
- Include 14
- Interaction Diagram 56
- Interaction Overview Diagram 65
- Interaktionsdiagramm 56
- Interaktionsübersichtsdiagramm 65
- Interfaces 46
- Jacobson 6
- Klasse 37
- Klassendiagramm 37
- Kollaborationsdiagramm 61
- Kommunikationsdiagramm 61
- Komponentendiagramm 66
- Komposition 42
- Kompositionsstrukturdiagramm 71
- Kontrollfluss 21
- Methoden 39
- Multiplizität 40
- Nachrichtenarten 56
- Object Diagram 72
- ObjectFlow 21
- Objekt 38
- Objektdiagramm 72
- Objektfluss 21
- offizielle Spezifikation
 - Normstelle 5
- OMG 6
- Operationen 39
- Package Diagram 53
- Paketdiagramm 53
- Parallelisierung 23
- Parametrisierbare Klassen 38
- realize 46
- Rollennamen 39
- Rumbaugh 6
- Schachteln von Aktivitätsdiagrammen 23
- Schnittstellen 46
- Scope 37
- Sequence Diagram 56
- Sequenzdiagramm 56
- Sichtbarkeitsbereich 37
- Spezialisierung 16, 44
- Splitting 23
- State Machine Diagram 32
- States 33
- Stereotype 38
- Strukturdiagramme 8
- Swimlanes 24
- Synchrone Nachrichten* 57
- Synchronisation 23
- Synchronisierungsbedingung, JoinSpec* 23
- System 13
- Timing Diagram 71
- Tokenkonzept für Aktivitätsdiagramme 21
- Transition 33
- UML Diagrammtypen 8
- Unterbrechungsbereich 25
- Use Case Diagram 11
- Verantwortlichkeitsbereiche 24
- Verhaltensdiagramme 8
- verlinkte Elemente 24
- Verlorene Nachrichten 57
- Verteilungsdiagramm 68
- Verzweigungen 22
- Zeitdiagramm 71
- Zusammenführen 22
- Zustände 33
- Zustandsdiagramm 32
- Zustandsübergang 33