Werner Schäfer

MASTER CLASS

Software-entwicklung

» Einstieg für Anspruchsvolle

inkl. Lerntest

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

3

UML

Information ist nur, was verstanden wird. (Card Friedrich von Weizsäcker)

Unified Modeling Language (UML) ist eine allgemein verwendbare, visuelle Modellierungssprache und Quasistandard für den Softwareentwurf. UML definiert eine Menge von grafischen Elementen, Beziehungen und Diagrammen. Es ist heute die gebräuchlichste Art, Modelle in der Softwareentwicklung zu erstellen. UML ist jedoch keine Methode, sondern eine Notation oder, wie der Name andeutet, eine Sprache. Als solche erleichtert sie wesentlich – oder sollte es zumindest – durch Anwendung einer einheitlichen Syntax die Kommunikation unter den Projektbeteiligten. UML beantwortet jedoch nicht die Frage, welche Diagramme für den jeweiligen Zweck sinnvoll und welche Schritte zu einem aussagekräftigen Softwareentwurf von Nöten sind, wie Musiknoten Regeln für das Niederschreiben einer Sinfonie festlegen, aber kein Rezept zum Komponieren einer Melodie beinhalten. Erst die Besprechung der einzelnen Disziplinen wie Architektur und Anforderungsanalyse wird Wege aufzeigen, UML angemessen und zweckdienlich einzusetzen.

Geschichte von UML

Doch vorerst zurück zu den Elementen von UML. Ein Blick in die kurze Geschichte der Modellierungssprachen zeigt, dass UML aus verschiedenen anderen Notationen hervorgegangen ist. Jede dieser Sprachen hatte ihre besondere Stärke und auch ihre eigene Anhängerschaft. Wie im Turmbau von Babylon wurde das Volk der Softwareentwickler in der Vergangenheit mit einer Sprachverwirrung bestraft. Fortan zierten Wolken, abgerundete Rechtecke, in sich geschachtelte Quadrate und Kreise die Wände der Entwicklungsbüros. Erst 1995 und später mit der Übernahme durch die *Object Management Group* (OMG) entstand aus den Methoden von Grady Booch, James Rumbaugh und Ivar Jacobson eine einheitliche Modellierungssprache. Der Funktionsumfang von der ersten Version 0.8 bis heute hat erheblich zugenommen und stellt nun einen De-facto-Standard in der Softwareentwicklung dar. UML taugt jedoch zu weit mehr als nur für die Modellierung von Software. So lässt sich UML in der Prozessanalyse wie auch in der Systemmodellierung einsetzen.

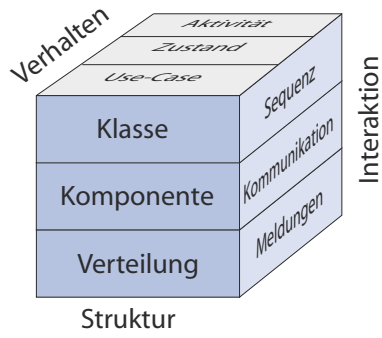
UML ist mit seinem stetig wachsenden Umfang bald selbst zum Problem geworden. Mit der aktuellen Version 2.2 kennt UML 14 verschiedene Diagramme, unterschiedlichste Objekte und Dutzende von Möglichkeiten, die Objekte untereinander zu verbinden. Hinzu kommt eine textuelle Sprache, um Bedingungen und Regeln zu formulieren. Zudem bestehen viele Möglichkeiten, die Sprache durch eigene Konstrukte zu erweitern. Die offizielle Spezifikation von UML von OMG umfasst inzwischen mehr als 700 Seiten! Dass unter solcher Geschäftigkeit von Normierungsgremien der praktische Bezug manchmal verloren geht, verwundert kaum. Statt jedoch UML akribisch einsetzen zu wollen, ist ein pragmatischer Umgang mit den Möglichkeiten dieser Sprache nötig.

Die folgenden Seiten verzichten bewusst auf Diagrammarten und Darstellungsformen, die außer einem akademischen Wert aus meiner Sicht keinen wirklichen Nutzen in der Softwaremodellierung haben. Dies ist auch keine Frage von plangetriebener oder agiler Vorgehensweise. Der maßvolle Einsatz von Mitteln gilt für alle gleichermaßen. Das, was von UML hier vorgestellt wird, ist und soll eine gemeinsame Sprache für alle Beteiligten am Softwareentwicklungsprozess sein.

3.1 UML-Würfel

UML besteht aus den drei Baugruppen: Dinge (Objekte), Beziehungen und Diagramme. Die Diagramme lassen sich weiter in die Kategorien Struktur, Verhalten und Interaktion aufteilen. Diese lassen sich auf die drei sichtbaren Seiten eines Würfels abbilden. Jede Seite teilt sich wiederum in drei Bereiche, die einen bestimmten Blickwinkel innerhalb der Kategorie einnehmen. Ein solcher Blickwinkel, oder wie hier als Perspektive bezeichnet, bedient sich eines spezifischen UML-Diagramms.

Abbildung 3.1
UML-Würfel



Struktur

Struktur ist eine Aufzählung der Bestandteile

Die Struktur beschreibt das Modell aus der statischen Sichtweise oder Perspektive. Es stellt dar, aus welchen Elementen und Bausteinen ein System besteht. Die Elemente in UML sind Klassen, Pakete, Komponenten und Kno-

ten. Die Klassen sind die kleinsten Teile, die zu Paketen zusammengefasst und dann zu Komponenten assembliert werden. Die Knoten stehen für Bauteile der Infrastruktur, auf denen die Komponenten während der Laufzeit ausgeführt werden. Die statische Sichtweise ist im Grunde eine Aufzählung und Gruppierung der Bestandteile eines Systems. Dabei werden Einzelbausteine schrittweise zu komplexeren Baugruppen komponiert. Die Struktur lässt sich durch die Darstellung der Klassen, Komponenten und die Verteilung auf die Knoten der Laufzeitumgebung dokumentieren. Diese Perspektive sagt jedoch nichts darüber aus, wie das System funktioniert oder es sich über die Zeit verhält. Mit der statischen Betrachtung werden die Prinzipien der Modularität, Abstraktion und Datenkapselung adressiert. Das Ziel ist es, weitgehend voneinander unabhängig operierende Bausteine zu erhalten.

Verhalten

Verhalten beschreibt die innere Dynamik

Das Verhalten dokumentiert den dynamischen Teil eines Systems. Es wird aufgezeigt, wie sich das System oder dessen Komponenten über die Zeit verhalten, um bestimmte Aufgaben zu erfüllen. Der Fokus liegt dabei auf einem bestimmten Baustein der statischen Perspektive. Die Elemente der dynamischen Perspektive sind Zustände, Aktivitäten, Meldungsflüsse bzw. Zustandsübergänge und Anwendungsfälle. Zustände und deren gleichnamige Diagramme veranschaulichen das Verhalten als ein Reagieren auf äußere Ereignisse aus einem stabilen Zustand heraus. Aktivitäten hingegen erzählen eine sequenzielle Abfolge von Tätigkeiten, die eine Komponente bis zu einem bestimmten Endpunkt ausführt. Anwendungsfälle schildern eine Abfolge von Interaktionen zwischen einem Benutzer und dem System. Oft werden Aktivitätsdiagramme auch für die Beschreibung von Anwendungsfällen verwendet. Dies ist nicht ganz unproblematisch, da Aktivitätsdiagramme dazu tendieren, das Verhalten als eine Folge von Systemaktivitäten und nicht als eine wechselseitige Interaktion darzulegen. Die Verhaltenssicht ist zudem der richtige Ort, um parallele Abläufe und Synchronisationspunkte innerhalb von Komponenten aufzuzeigen.

Interaktion

Interaktion zeigt den Meldungsfluss über mehrere Komponenten

Die Interaktionen sehen das System als einen Fluss von Meldungen. Es wird dargelegt, welche Meldungen zwischen den statischen Elementen fließen und wie diese zeitlich zusammenhängen. Im Gegensatz zur Verhaltensperspektive liegt der Schwerpunkt auf dem Zusammenspiel der Komponenten von außen gesehen. Betrachtete die dynamische Sichtweise das Verhalten eines einzelnen Bausteins oder eines Subsystems unabhängig von den anderen, so zeigt die Interaktionsperspektive den dabei stattfindenden Meldungsaustausch zwischen diesen Bausteinen auf. Für die Darstellung der Interaktion werden Sequenz- und Kommunikationsdiagramme genutzt. Beide zeigen den zeitlichen Meldungsverlauf auf und illustrieren, welche Spur die Verarbeitung eines bestimmten Ereignisses im betrachteten Systemabschnitt hinterlässt. Mit der Beschreibung der Interaktionen lassen sich komplexe Zusammenhänge zwischen den Bausteinen jeder Abstraktionsebene ausdrücken. So lässt sich die Reaktion des Systems auf einzelne Szenarien illustrieren, indem

der Pfad des Meldungsverlaufs aufgezeigt wird. Die Mächtigkeit der Interaktionsdiagramme soll nicht darüber hinwegtäuschen, dass diese oft schwer lesbar und noch schwerer wartbar sind. Sequenzdiagramme sollen bewusst einfach gehalten und nur für das Aufzeigen der wichtigsten Kommunikationspfade verwendet werden. Ist deren Granularität zu fein, sind sie gerade in der Implementierungsphase ständigen Änderungen unterworfen.

Diese drei Perspektiven beschreiben mit ihren jeweiligen Dingen, Beziehungen und Diagrammen das Modell von verschiedenen Seiten. Eine statische Struktur ist nur ein Teil davon und hilft für sich alleine wenig, um ein System verstehen zu können. Es braucht ein ausgewogenes Verhältnis aller Sichtweisen. Gerne konzentrieren sich Modelle auf den strukturellen Teil, da dieser weniger durch Änderungen betroffen ist und oft eine geringere Komplexität aufweist als die anderen Perspektiven. Doch erst die Modellierung des dynamischen Verhaltens und das Aufzeigen des Meldungsverlaufs verleiht dem Modell Leben und erklärt dessen Funktionsweise. Ein Unternehmen lässt sich auch nicht nur mittels Organigramm erklären. Es bedarf der Geschäftsprozesse, um zu verstehen, wie eine Firma „funktioniert".

Sichten, Standpunkte und Perspektiven

Sichten, Standpunkte und Perspektiven? Diese Begriffe werden in der Literatur nicht einheitlich verwendet. Nach IEEE-Standard ist ein Standpunkt eine Menge von Mustern, Vorlagen und Regeln, welche das Modell aus einem bestimmten Blickwinkel beschreibt. Eine Sicht ist hingegen das, was jeweils von einem bestimmten System aus einem generellen Standpunkt gesehen wird. Jeder Standpunkt bedient sich zur Beschreibung einer Sicht der drei Perspektiven Struktur, Verhalten und Interaktion in unterschiedlicher Ausprägung. So lässt sich eine Software aus dem Blickwinkel des funktionalen Standpunkts erläutern, indem die Struktur der Funktionsblöcke aus der statischen Perspektive beschrieben wird.

3.2 Struktur

3.2.1 Klassendiagramm

Klassen repräsentieren verschiedene Dinge

Ausgangslage jedes Modells in UML ist das Klassendiagramm. Es zeigt die einzelnen Bausteine der Software, deren Attribute, Operationen und Beziehungen untereinander. Was genau eine Klasse ist, hängt von der jeweiligen Betrachtungsebene ab und ist nicht auf die Darstellung von Softwarebausteinen eines ausführbaren Programms beschränkt. Auf Unternehmensebene kann mit einer Klasse ein Server gemeint sein, der bestimmte Dienste als Services bereitstellt und durch bestimmte Leitungsmerkmale charakterisiert ist. Diese Flexibilität erlaubt es, UML für die unterschiedlichsten Aufgaben einzusetzen: vom Entwurf eines Softwareprogramms bis zur Beschreibung von Geschäftsprozessen und Unternehmensstrukturen. Doch vorerst zurück zu den Grundlagen eines Klassendiagramms.

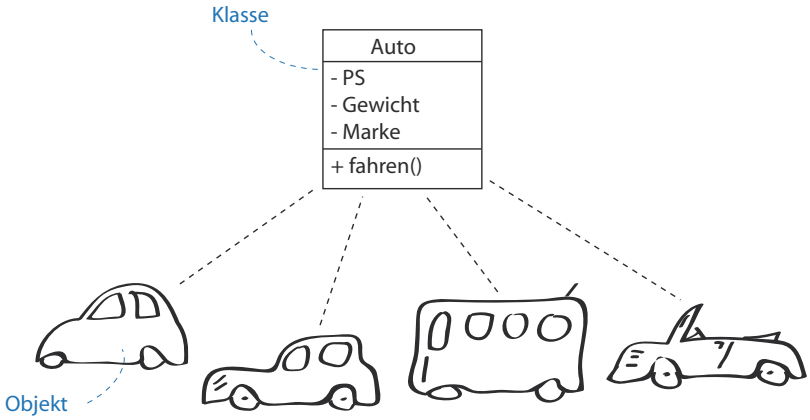


Abbildung 3.2
Eine Klasse ist eine Abstraktion.

Klassen fassen Gemeinsamkeiten zusammen

Ein Klassendiagramm besteht aus den Dingen *Klassen*, *Pakete* und *Objekte* sowie den Verbindungen *Assoziationen*, *Kompositionen* und *Vererbungen*. Eine Klasse repräsentiert und abstrahiert Gemeinsamkeiten von Gegenständen. Sind Objekte konkrete Instanzen realer, abzählbarer Gegenstände, so beschreibt die Klasse die gemeinsamen Eigenschaften sowie Verantwortlichkeiten der gleichartigen Objekte. Die Klasse ist eine Schablone, die ein gemeinsames Verhalten kennzeichnet. Was eine Klasse ist, hängt dabei vom Betrachter und dem jeweiligen Systemzweck ab. So sind beispielsweise Ehemann Peter, Tochter Sue und ihr Hund Waldo aus Sicht eines Transportunternehmens Fahrgäste, jedoch nur Peter und Sue sind auch Kunden. Klassen – wie es der Name sagt – klassifizieren Dinge der wirklichen Welt in einem für den Zweck entsprechenden Ordnungssystem: im vorherigen Beispiel in Kunden und Fahrgäste. Dabei charakterisieren die Attribute, deren Ausprägung und die Operationen, die durch die Instanzen einer Klasse ausführbaren Aktionen oder Tätigkeiten.

Klasse

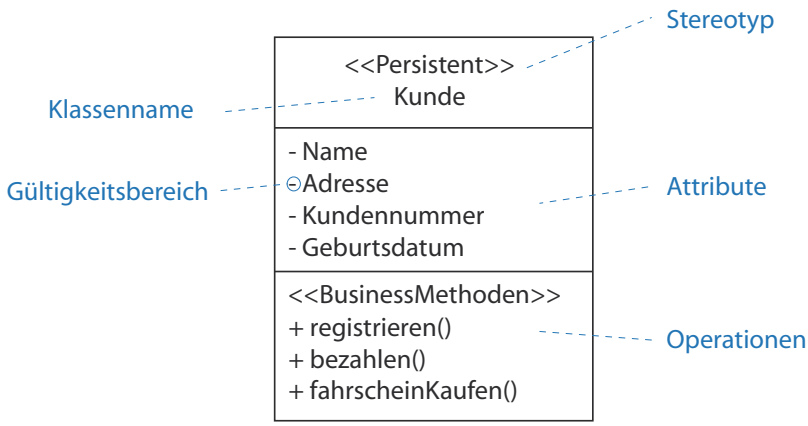


Abbildung 3.3
Klassenelemente

Eine Klasse besteht aus folgenden Bestandteilen:

- *Klassenname*
  Jede Klasse unterscheidet sich eindeutig durch ihren Namen von allen anderen. Der Klassenname besteht aus dem eigentlichen Namen und einem vorangestellten Pfad oder Namensraum (qualifizierter Name). Pakete – in UML durch eine Mappe symbolisiert – teilen den Namensraum wie eine Verzeichnisstruktur hierarchisch auf. Dies erlaubt es, denselben Namen in unterschiedlichen Kontexten zu verwenden. Im qualifizierten Namen werden die einzelnen Verzeichnisse oder Pakete durch zwei Doppelpunkte voneinander getrennt. Zum Beispiel: `M-Ticket::BusinessObjekte::Kunde`.
- *Stereotypen*
  Stereotypen sind formale Erweiterungen einer Klasse, indem sie dieser eine zusätzliche Semantik verleihen. Damit können Klassen um spezifische Aspekte erweitert werden, die ein bestimmtes Verhalten implizieren. So definiert der Stereotyp *Interface*, dass die Klasse eine Schnittstelle spezifiziert, welche durch andere Klassen zu implementieren ist.
- *Attribute*
  Attribute beschreiben Eigenschaften einer Klasse, die in der jeweiligen Instanz, bestimmte Werte annehmen können. Attribute sind Informationen eines bestimmten Datentyps, der je nach Abstraktionsgrad unterdrückt oder hinter dem Attributnamen getrennt durch einen Doppelpunkt angezeigt wird: beispielsweise: `Name:String`.
- *Operationen*
  Operationen implementieren einen bestimmten, durch die Klasse bereitgestellten Service. Eine Operation ist dabei eine Abstraktion von etwas, das ein Objekt tun kann und das allen Objekten dieser Klasse gemein ist. So besagt die Abbildung 3.3, dass alle Objekte der Klasse *Kunden* Fahrscheine kaufen können. Durch Stereotypen lassen sich Attribute und Operationen gruppieren. So kennzeichnet der Stereotyp `BusinessMethoden` alle nachfolgenden Operationen als Geschäftsfunktionen.
- *Gültigkeitsbereich*
  Die Symbole –, + und ~ vor den Attributen und Operationen definieren deren Gültigkeitsbereich. Das Minuszeichen steht für *private,* also für eine Gültigkeit innerhalb dieser Klasse. Das Pluszeichen steht für *public*, also für alle sichtbar. Das Tildenzeichen für *protected* bedeutet, dass das Element für alle davon abgeleiteten Klassen sichtbar ist. Wird die Gültigkeit nicht explizit ausgewiesen, so bedeutet dies für Attribute private und Operationen öffentlich.

Diagrammgröße

Je nach Detaillierungsgrad eines Diagramms werden Elemente bewusst weggelassen. Liegt das Interesse auf der Struktur und den Beziehungen, werden die Operationen und zum Teil auch die Attribute unterdrückt. Damit gewinnt ein Diagramm an Übersicht. An anderer Stelle wird dann der Fokus auf die Einzelheiten einer Klasse gelegt und dafür die Umgebung ausgeblendet. Es empfiehlt sich, unterschiedliche Klassendiagramme zu verwenden und nicht der Versuchung zu erliegen, das gesamte System in einem einzigen, riesigen

Plakat unterbringen zu wollen. Als Faustregel gilt: Die Darstellung sollte vollständig und dabei noch lesbar auf einer Bildschirmseite eines 20-Zoll-Monitors Platz finden.

Beziehungen

Assoziationen geben strukturelle Abhängigkeiten wieder

Klassen stehen natürlich nicht für sich allein, sondern haben vielfältige Beziehungen untereinander. Diese lassen sich in die Kategorien Assoziation, Abhängigkeit, Komposition und Vererbung unterteilen. Assoziationen sind strukturelle Beziehungen beliebiger Art und verbinden zwei Klassen miteinander. Die Multiplizität solcher Assoziationen sagt etwas darüber aus, in welchem zahlenmäßigen Verhältnis die Klassen zueinander stehen. Die mit einer Pfeilspitze versehene Bezeichnung gibt die Lesart der Verbindung wieder. So sagt in der Abbildung 3.4 die Beziehung zwischen Kunde und Ticket aus, dass ein Kunde keine oder mehrere Tickets gekauft hat.

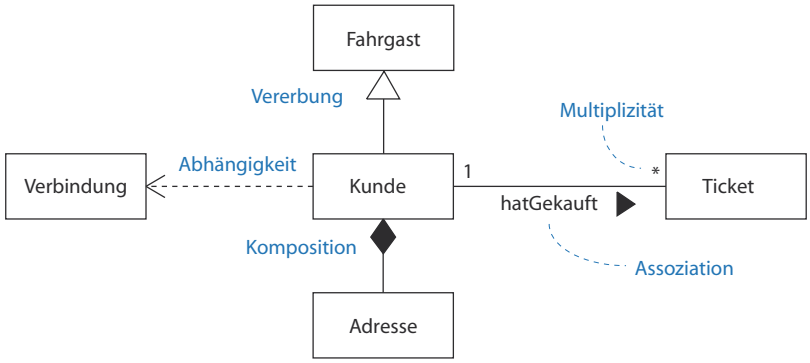


Abbildung 3.4
Klassenbeziehungen

Die durch eine gestrichelte Linie wiedergegebene Abhängigkeit legt dar, dass eine Klasse eine andere braucht (use-dependency). Dies kann als Parameter einer Operation sein oder für deren Implementation. So nutzt ein Kunde in der Abbildung 3.4 eine bestimmte Bahnverbindung, es besteht aber keine strukturelle Abhängigkeit.

Vererbung fasst Gemeinsamkeiten von Klassen zusammen

Die Vererbung markiert, dass eine Klasse von einer allgemeineren Basisklasse abgeleitet wurde. Dabei erbt die Klasse die Eigenschaften und das Verhalten ihrer Basisklasse. Mit dem Vererbungspfeil wird eine Generalisierung angezeigt. Beispielswiese ist der Fahrgast eine Generalisierung des Kunden. Jeder Fahrgast kann mit dem Zug reisen, doch nur Kunden sind in der Lage, Fahrkarten zu kaufen. Entgegen der Bezeichnung Vererbung handelt es sich dabei nicht um eine Vererbung im herkömmlichen Sinne. Die Basisklasse befindet sich lediglich auf einer höheren Abstraktionsstufe als die davon abgeleitete Klasse selbst. So teilen Menschen und Elefanten die Eigenart von Säugetieren, aber nicht deren Gene. Die Blutsverwandtschaft zwischen Vater und Tochter wird hingegen durch eine Assoziation angedeutet. Stellen Klassen Gemeinsamkeiten von Objekten dar, so fassen Basisklassen die Gemeinsamkeiten von Klassen zusammen. Die Von-Neumann-Rechnerarchitektur, um

ein anderes Beispiel anzuführen, ist ein Referenzmodell heutiger Computer und damit Basisklasse von PC und Server gleichermaßen.

Komposition beschreibt eine Teil-Ganzes-Beziehung

Der letzte Typ von Beziehungen sind Kompositionen. Diese sind Verbindungen, die ein Ganzes bestehend aus seinen Teilen beschreiben. Repräsentieren die Assoziationen strukturelle Bindungen auf derselben konzeptionellen Ebene, so zeigen Kompositionen, dass ein Ding ein Element eines übergeordneten, größeren Ganzen ist. Diese Verbindungsart impliziert normalerweise, das Ganze „besitze“ seine Teile. Exemplarisch sei hier ein Transistorradio aufgeführt. Es setzt sich vereinfacht ausgedrückt aus Empfänger, Verstärker und Lautsprecher zusammen. UML kennt neben der Komposition die etwas schwächere Form der Aggregation. Die Unterscheidung ist jedoch etwas realitätsfremd. Entweder ein Ding ist Teil einer Sache oder es besteht eine Assoziation auf gleicher Ebene. Deshalb verwenden wir fortan nur die Komposition, symbolisiert durch eine ausgefüllte Route.

Objekte

Ein Objekt hat einen Zustand, ein bestimmtes Verhalten und eine eindeutige Identifikation

Ein Objekt oder als Synonym dazu die Instanz ist ein konkretes Ding der realen Welt. Es leitet sich von einer Abstraktion dessen ab, was wir als Klasse bezeichnet hatten. Faktisch ist dies für ein Modell nicht ganz richtig. Dieses Objekt ist, um es genau zu nehmen, selbst eine Abstraktion der zu modellierenden Welt. Beispielsweise ist die Instanz Peter im System M-Ticket eine auf den Kundenaspekt reduzierte Darstellung einer Person mit dem Namen Peter. Ein Objekt hat eine eindeutige Identifikation, einen Zustand und ein von der Klasse vorgegebenes Verhalten. Der momentane Zustand eines Objekts manifestiert sich durch die jeweiligen Werte der Klassenattribute. Objektdiagramme, wie in der Abbildung 3.5 dargestellt, sind eher die Ausnahme in der Modellierung. Jedoch werden sie in den später besprochenen Kommunikationsdiagrammen eine praktische Anwendung finden.

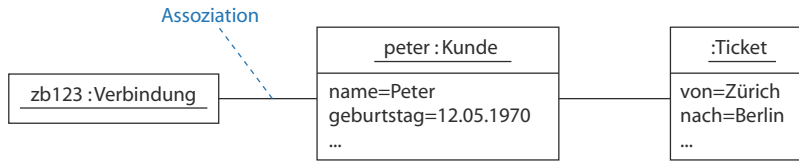


Abbildung 3.5
Objektdiagramm

3.2.2 Komponentendiagramm

Komponenten stellen bestimmte Leistungen über wohldefinierte Schnittstellen bereit

Eine Komponente ist ein von anderen Teilen unabhängig liefer- und verteilbarer Funktionsblock mit einem über Schnittstellen definierten Zugriff auf dessen Services. Oder anders ausgedrückt, eine Komponente stellt über Schnittstellen eine bestimmte Funktionalität bereit. Eine Schnittstelle ist dabei der Vertrag zwischen Leistungserbringer, der Komponente und dem Leistungsnutzer. Dieser Vertrag regelt die Bedingungen, unter denen ein bestimmter Dienst erbracht wird, und wie die Leistung bzw. deren Ergebnis aussieht. Vergleichbar mit einem Service-Level-Agreement (SLA) im Dienstleistungsbereich, definiert der Vertrag nur die zu erbringende Leistung, legt

aber nicht fest, wie diese zu erbringen ist. Solange der Nutzer einer Komponente nur von dieser Übereinkunft und nicht von der konkreten Implementierung abhängt, lässt sich der Baustein jederzeit durch eine andere Komponente mit denselben Schnittstellen ersetzen. Komponenten sind damit im Unterschied zu den Klassen während der Laufzeit austauschbar und deren Implementierung bleibt hinter den Schnittstellen verborgen. Der Wunsch nach universellen, frei einsetzbaren Softwarebausteinen wird nur durch unterschiedliche Laufzeitumgebungen getrübt. So ist eine EJB-Komponente nicht ohne Weiteres in der Laufzeitumgebung von CORBA einsetzbar.

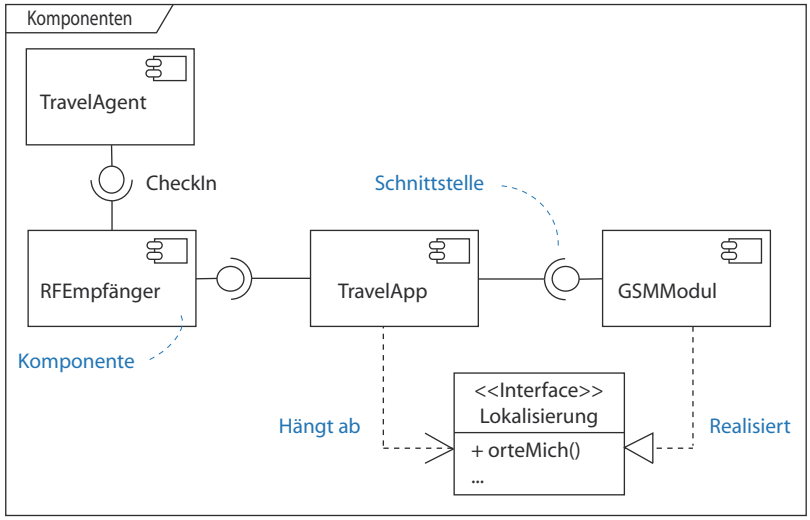


Abbildung 3.6
Komponentendiagramm

Komponente oder Klasse

Doch was stellt eine Komponente in UML dar und wie unterscheidet sich diese von einer Klasse? Komponenten implementieren, wie es Klassen auch tun können, ein oder mehrere Schnittstellen. Sie weisen jedoch keine weiteren Operationen oder Attribute außerhalb dieser durch die Schnittstellen bereitgestellten Operationen auf. Klassen sind zudem vererbbar, Komponenten nicht. Komponenten repräsentieren physikalische Dinge oder die Paktierung von logischen Elementen zu verteilbaren Einheiten. Sie stellen eine in sich geschlossene Funktionalität dar, wie beispielsweise XML Parser, Kreditkartenmodule für Online-Bezahlung oder einen Verschlüsselungsdienst. Bei vielen als Komponenten bezeichneten Bausteinen handelt es sich jedoch um Klassen. Java-Komponenten in EJB oder .NET-Komponenten der Enterprise Services sind im eigentlichen Sinne Klassen.

Interface-Klassen definieren die über eine Schnittstelle verfügbaren Funktionen

Die für Komponenten – wie auch Klassen – über Schnittstellen zur Verfügung gestellten Services werden als Lollipop-Symbol gekennzeichnet. Dem gegenüber wird die Nutzung dieser angebotenen Leistungen durch eine Gabel symbolisiert. Diese Darstellung verbirgt, dass die Schnittstelle durch eine gemeinsame Interface-Klasse spezifiziert ist. Eine solche Interface-Klasse besteht nur aus der Definition von Operationen, die durch die Komponente implementiert werden. Die Interface-Klasse „Lokalisierung“ in Abbildung 3.6 zeigt eine

andere Form der Darstellung anstelle des Lollipop-Symbols. Die gestrichelte Verbindung mit einer geschlossenen Pfeilspitze gibt an, dass die Komponente den durch die Schnittstelle spezifizierten Vertrag realisiert bzw. implementiert; die gestrichelte Verbindung mit der offenen Pfeilspitze beschreibt, dass eine Komponente von einer Schnittstelle abhängt.

Kehren wir zurück zu unserem Beispiel in Abbildung 3.6 und stellen fest, dass die Erkennung eines mobilen Geräts an den Ein- und Ausstiegspunkten eine Komponente ist, die wir beabsichtigen, käuflich zu erwerben. Diese Komponente soll dem Standard NFC genügen, leicht einzubauen sein und eine Zuverlässigkeit von 99,9 % aufweisen. Fortan bezieht sich der weitere Softwareentwurf auf die Spezifikation einer solchen Schnittstelle. Die Architektur hat zu verifizieren, ob eine solche Lösung machbar und in absehbarer Zeit verfügbar ist. Eine Standardkomponente für die Ortung des Mobiltelefons sorgt zudem für die Bestimmung der gefahrenen Route. Der Ein- und Ausstieg wird durch den stationären Empfänger an die Zentrale übermittelt. Die Komponente *TravelApp* kann nun basierend auf den Spezifikationen dieser Schnittstellen entwickelt werden, ohne das definitive Produkt bereits zu kennen.

3.2.3 Kompositionsdiagramm

Kompositionsdiagramme ermöglichen Dekomposition

UML 2 führte mit der Komposition (Composition Structure) ein neues Diagramm ein, welches es erlaubt, die innere Struktur einer Klasse oder Komponente darzustellen. Damit ist es möglich, eine hierarchische Dekomposition in UML zu modellieren. Die Schöpfer von UML beabsichtigten, eine einheitliche Sprache für objektorientierte Programmierung zu schaffen. Idealisiert besteht diese Welt aus eigenständigen Objekten, die sich zu Komponenten und Systemen formieren. Eine Dekomposition, also die Zerlegung einer Klasse in Unterklassen wie aus einer datenorientierten Modellierung bekannt, war nicht vorgesehen. Mit Kompositionsdiagrammen ist es nun auch in UML möglich, die innere Struktur von Klassen und Komponenten darzustellen.

Ein Kompositionsdiagramm besteht aus Anschlüssen, Teilen und Konnektoren. Die inneren Elemente werden Teile genannt, welche über Anschlüsse und Konnektoren miteinander verbunden sind und über die Schnittstellen der umschließenden Klassen oder Komponente mit der Außenwelt verknüpft werden. Die Anschlüsse, symbolisiert als kleine Quadrate, sind logische oder physische Schnittstellen, über die Services erbracht bzw. gefordert werden. Die Konnektoren zwischen den Anschlüssen oder Teile einer Klasse zeigen auf, über welche physikalischen oder logischen Verbindungen die Kommunikation stattfindet. Wie die Abbildung 3.7 illustriert, ist es nicht zwingend erforderlich, den Anschluss (Port) anzugeben, wenn daran keine bereitgestellten oder genutzten Schnittstellen geknüpft werden.

Die Kompositionsstrukturen wurden im UML-Derivat SysML als interne Blockdiagramme erweitert und verbessert. Einige dieser Erweiterungen halten nun umgekehrt auch Einzug in die UML-Spezifikation. So wird jetzt

eine Klasse, welche nicht direkt Teil der übergeordneten Klasse ist, im Kompositionsdiagramm als Baustein bzw. Property bezeichnet und durch eine gestrichelte Umrandung hervorgehoben. Im Beispiel verwendet ein Handy die Komponente *TravelApp,* ohne jedoch ein fixer Bestandteil eines mobilen Telefons zu sein.

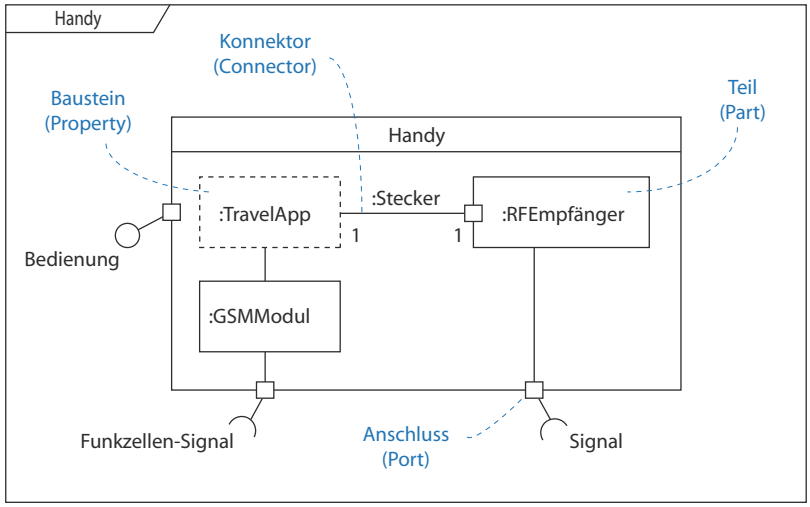


Abbildung 3.7
Kompositionsdiagramm

Kompositionsdiagramme für die strukturelle Architektur

In der Praxis eignet sich das Kompositionsdiagramm speziell für die Beschreibung von Systemarchitekturen auf Unternehmens- und Applikationsebene, also dort, wo physikalische Komponenten zu Subsystemen und Applikationsverbunden zusammengefasst werden und das Interesse auf dem Meldungsfluss und den Schnittstellen liegt. Im Gegensatz zur Darstellung der Verteilung blickt die Komposition aus konzeptioneller Sicht auf das System. Hier liefert sie den statischen Entwurf für die Schilderung des Interaktionspfads, ohne bereits die konkrete Laufzeitumgebung nennen zu müssen. Kompositionen sind *das* mentale Modell für das von Peter Hruschka geforderte Schwarz-Weiß-Denken. Auf einer höheren Abstraktionsebene werden Komponenten oder Subsysteme als Blackbox betrachtet. Nur das von außen feststellbare – durch die Spezifikation der Schnittstellen definierte – Verhalten ist von Bedeutung. Deren Implementierung interessiert erst auf einer tieferen Ebene. Bei dieser Whitebox-Betrachtung richten wir den Blick auf das Innenleben der Komponente.

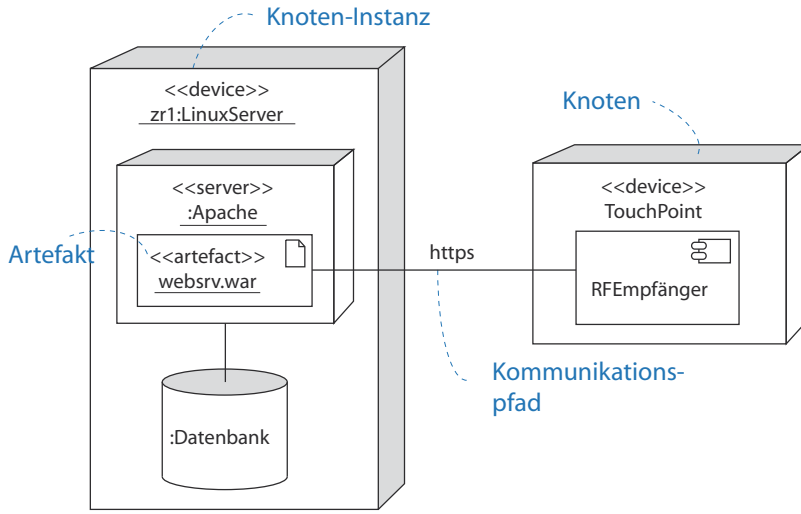
3.2.4 Verteilungsdiagramm

Verteilung der Komponenten auf Knoten

Das Verteilungsdiagramm zeigt, welcher Teil der Software auf welcher Hardware oder Ausführungseinheit läuft. Es legt die physikalische Architektur bestehend aus Infrastruktur und deren Kommunikationspfaden fest. Die Hardware und Ausführungseinheiten werden durch sogenannte Knoten repräsentiert, welche ineinander verschachtelt sein können. Dabei stellt die Hardware ein physikalisches Gerät wie einen PC oder Server dar und die Aus-

führungseinheiten ein Stück Software wie Betriebssysteme, Applikationsserver, Datenbanksysteme und Webserver. Knoten können wie Klassen abstrakte Einheiten abbilden, beispielsweise einen mit Windows installierten Computer oder einen Linux-Server. Als konkrete Objekte, gekennzeichnet durch einen unterstrichenen Namen, stellen sie eine bestimmte Instanz eines physischen Rechners dar. Die Darstellung des Knotens als Box ist nicht zwingend. So kann für eine bessere Verständigung der Knoten einer Datenbank als Zylinder visualisiert werden.

Abbildung 3.8
Verteilungsdiagramm



Artefakte

Artefakte repräsentieren die Spezifikation realer Dinge wie Dateien, Bibliotheken, ausführbare Programme, Dokumente, Konfigurationsdateien oder Datenbanktabellen. Als Objekte dargestellt (Name ist unterstrichen), stehen sie für eine spezifische Instanz eines Artefakts. Ein Artefakt ermöglicht die physische Manifestation eines beliebigen Elements von UML, typischerweise Komponenten oder Dateien. In der Praxis wird nicht immer formal richtig zwischen Instanz und Klasse von Knoten und Artefakten unterschieden.

3.3 Verhalten

3.3.1 Aktivitätsdiagramm

Aktivitätsdiagramme sind erweiterte Flussdiagramme

Aktivitätsdiagramme sind im Grunde erweiterte, objektorientierte Flussdiagramme. Sie beschreiben prozedurale Logik, Geschäftsprozesse oder Arbeitsabläufe. Im Unterschied zu den einfachen Flussdiagrammen erlauben Aktivitätsdiagramme die Darstellung von parallelen Abläufen. Mit UML 2 wurden viele Änderungen eingeführt und die Semantik wurde an den Formalismus der Petri-Netze angeglichen. Dazu gehört auch eine klare Abgrenzung von den Zustandsdiagrammen. Aktivitätsdiagramme stellen das dynamische Verhalten eines Bausteins oder einer Operation durch die Sequenz von ein-

zelnen Aktionen dar. Die Aktionen lassen sich zu Aktivitäten und diese zu übergeordneten Abläufen zusammenfassen. In umgekehrter Richtung lässt sich damit ein komplexes Verhalten schrittweise in überblickbare Tätigkeiten aufteilen: Eine Operation besteht aus einer Abfolge mehrerer Aktivitäten und jede Aktivität selbst wieder aus mehreren atomaren Aktionen.

Aktivitätsdiagramme bestehen aus Aktionen, Kontrollelementen und Objekten

Die Symbole eines Aktivitätsdiagramms lassen sich in Aktionen, Kontrollelemente und Objekte aufteilen. Aktionen repräsentieren eine diskrete, nicht weiter aufteilbare Arbeitseinheit. Kontrolleinheiten steuern den Ablauf durch Entscheidungs- und Zusammenführungspunkte, Teilungs- und Synchronisationsbalken sowie Start- und Endpunkt. Objekte bezeichnen Instanzen von Klassen, die durch eine Tätigkeit erzeugt oder konsumiert werden. Die gerichteten Verbindungen symbolisieren den Fluss einer Sequenz oder eines Objekts. Sobald an allen Eingangsflüssen Objekte anliegen, wird die Aktion oder Aktivität ausgeführt. Eine Aktivität lässt sich durch vertikale oder horizontale Partitionen in Verantwortungsbereiche auftrennen. Partitionen können die unterschiedlichsten Dinge repräsentieren: Klassen, Komponenten, physische Knoten, Organisationseinheiten oder Rollen. Grundsätzlich haben Partitionen, außer einer Verbesserung der Lesbarkeit, keine weitere Bedeutung. Früher wurden diese Partitionen Swimlanes genannt.

Aktionen

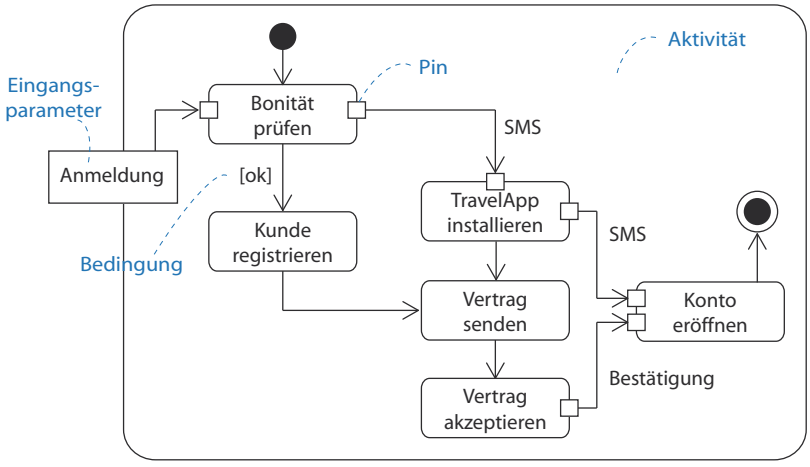


Abbildung 3.9
Aktivitäten und Aktionen

Aktivitäten

Die Aktivität ist eine in sich geschlossene Tätigkeit mit einem definierten Ergebnis. Eine solche Aktivität stellt einen Teilprozess oder eine Operation dar, welche weiter in Einzelschritten aufteilbar ist. Eine solche Aktivität hat einen eigenen Start- und Endpunkt. Oft werden die Aktivitäten über deren Arbeitsergebnis miteinander über sogenannte Objektflüsse miteinander verbunden. Dabei symbolisiert ein Pin oder Eingangsparameter, dargestellt als kleines Rechteck, den Eintritt eines Objektflusses. Aktivitäten können zudem Vor- und Nachbedingungen aufweisen. So kann die Vorbedingung für die Aktivität „Einchecken“ im Beispiel eine vorhandene Registrierung und die

Nachbedingung die Eröffnung der Fahrt auf dem Server sein. Wie wir später noch sehen werden, sind Vor- und Nachbedingungen ein nützliches Mittel zur Spezifikation von Operationen.

Aktionen

Eine Aktion ist eine nicht weiter teilbare, atomare Tätigkeit. Aktionen werden dann ausgeführt, wenn an allen Eingangskontrollflüssen oder Objektflüssen ein Wert (Token) anliegt. Es ist eine UND-Verknüpfung. Nach der vollbrachten Aktion liegt an allen Ausgangskontrollflüssen ein Wert an. Kontrollflüsse können mit Bedingungen versehen werden, die in eckigen Klammern darzustellen sind. Eine solche Bedingung funktioniert wie ein Filter und lässt nur jene Tokens durch, die diese Vorgaben erfüllen. Abbildung 3.10 illustriert dieses Verhalten: Die Aktion ist erst ausführbar, wenn an allen Eingängen ein Wert anliegt. Nach Ausführung der Aktion liegt dann an allen Ausgängen genau ein Wert. Dieses Verhalten entspricht dem der Petri-Netze.

Abbildung 3.10
Aktionsausführung

Objekte

Objekte

Objekte stellen konkrete Arbeitsergebnisse einer Aktivität oder einer einzelnen Aktion dar. Dabei ist ein Objektknoten, dargestellt als Rechteck, ein Datenspeicher, der ohne spezielle Anmerkung unendliche Kapazität aufweist und nach dem FIFO (First-In-Last-Out)-Prinzip funktioniert. Ein Objekt dieses Datenspeichers wird an allen Ausgängen gleichzeitig angeboten, jedoch nur exakt eine Aktion kann dasselbe Objekt konsumieren. In einer verkürzten Version wird der Objektfluss durch Ein- und Austrittspins an den jeweiligen Aktivitäten symbolisiert. Das Objekt selbst, welches über diese Verbindung fließt, wird nicht mehr explizit dargestellt.

Kontrollelemente

Entscheidungspunkte

Entscheidungspunkte visualisiert als Raute sind einfache „Wenn-dann"-Bedingungen. Jene ausgehende Kante, welche die Bedingung erfüllt, wird durch das Token traversiert, das heißt, es folgt diesem Kontrollfluss. Es kann aber nur jeweils eine dieser Kanten beschritten werden. Sollten mehrere abgehende Kanten den Bedingungen genügen, so ist der weitere Verlauf nicht bestimmbar. Das bedeutet, es ist nicht definiert, welche Kante das Token wählt. Eine mögliche Bedingung wird auch hier in eckigen Klammern dargestellt. Mit dem Schlüsselwort *else* wird jene Kante gekennzeichnet, welche gewählt wird, wenn alle anderen Bedingungen nicht wahr sind. Diagramme mit vielen Entscheidungspunkten werden schnell unübersichtlich. Statt eines Entscheidungspunkts können die ausgehenden Kanten mit entsprechenden Bedingungen direkt zwischen zwei Aktionen verbunden werden. So sind auch mehrfach hintereinander geschaltete Entscheidungspunkte durch eine kombinierbare Bedingung wesentlich einfacher und übersichtlicher darstellbar. Für ungeübte Leser von UML ist jedoch die klassische

Darstellung mit Entscheidungspunkten leichter verständlich als boolesche Ausdrücke.

Verbindungspunkte

Verbindungspunkte sehen aus wie verkehrte Entscheidungspunkte. Deren Semantik ist denkbar einfach. Sie führen mehre Eingangsflüsse zu einem Ausgangsfluss zusammen, ohne den Wert eines Tokens zu ändern. Ein Verbindungspunkt verhält sich wie die Verschmelzung von zwei Flussläufen. Ein Verbindungspunkt lässt sich mit einem nachfolgenden Entscheidungspunkt kombinieren. So entsteht ein Entscheidungspunkt mit mehreren Eingangsflüssen.

Synchronisation

Synchronisationsbalken werden als vertikaler oder horizontaler Balken dargestellt. Erst wenn an allen Eingangsflüssen ein Wert anliegt, ist die Bedingung für die ausgehende Kante erfüllt. Der Synchronisationsbalken ist eine UND-Verknüpfung und vereint alle Eingangswerte zu einem einzigen Ausgangswert. Eine Synchronisation kann über die einfache Verknüpfung hinaus durch einen booleschen Ausdruck exakter spezifiziert werden. So kann der Synchronisationsbalken der Abbildung 3.11 mit der Bedingung versehen werden, dass beim Auschecken der Zielbahnhof nicht gleich dem Startbahnhof sein darf. Wie an anderen Orten auch werden Bedingungen in Form von booleschen Ausdrücken durch geschweifte Klammern markiert.

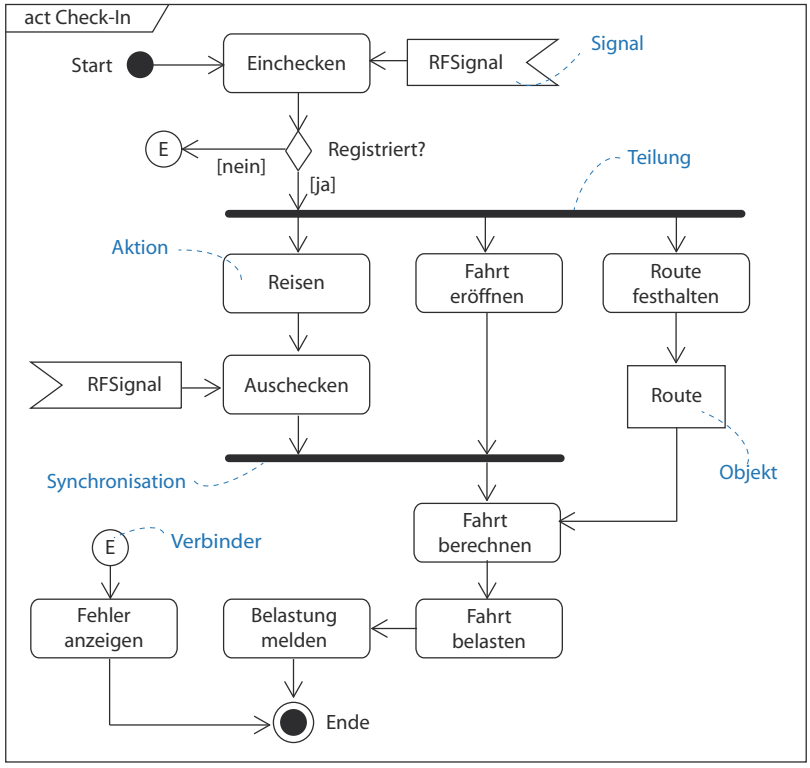


Abbildung 3.11
Kontrollelemente

Teilung

Der Teilungsbalken ist das Gegenteil des Synchronisationsbalkens. Alle Werte bzw. Tokens eines Eingangsflusses werden dupliziert, so dass am Ende jeder Ausgangsfluss einen Wert erhält. Dabei verzweigt der weitere Ablauf auf zwei oder mehrere parallele, voneinander unabhängige, Ausführungsstränge. Ein solcher Strang repräsentiert sich in der Regel als eigenständiger Prozess oder Thread. Die Teilung ermöglicht die Darstellung von gleichzeitig stattfindenden Aktivitäten.

Signale

Signale werden dazu eingesetzt, auf asynchrone, von außen kommende Ereignisse zu warten oder ein solches Ereignis auszulösen. Ein eingebuchtetes Fähnchen stellt das Warten auf ein Ereignis; das ausgebuchtete das Versenden eines Ereignisses dar. So wartet im Beispiel der Abbildung 3.11 die Aktion Ein- und Auschecken auf das Signal vom RF-Empfänger des Touch-Points. Mit einem Signal kann auch das Erreichen eines bestimmten Zeitpunkts wie das Ende des Monats modelliert werden.

Aktivitätsdiagramme weisen noch weitere Elemente und Spezialitäten auf, welche mit jeder Version von UML zunehmen. Damit lässt sich nahezu jedes dynamische Verhalten darstellen. Der Versuch, jede noch so ausgefallene Situation zu modellieren und dabei eine Vielzahl von Symbolen einzusetzen, endet zumeist in einer Überfrachtung der Modelle. Zudem bedürfen solche selten verwendete Symbole oft einer Erklärung. Mit den zuvor aufgezählten Elementen lassen sich die meisten Situationen noch einigermaßen verständlich darstellen. Die zunehmende Komplexität der Aktivitätsdiagramme war sicher einer der Gründe, dass die Wirtschaftsinformatik mit der BPM-Notation zurück zu einfacheren Ablaufbeschreibungen gefunden hat. Trotzdem sind Aktivitätsdiagramme vielfältig einsetzbar und werden im Allgemeinen durch ein breites Publikum gut verstanden. Mit Aktivitäten lassen sich Geschäftsprozesse genauso wie die Ausführung einer Operation auf Codeebene erklären. Aktivitätsdiagramme gehören ohne Zweifel zu den wirklich praktischen Dingen von UML, auch außerhalb der Softwareentwicklung.

3.3.2 Zustandsdiagramm

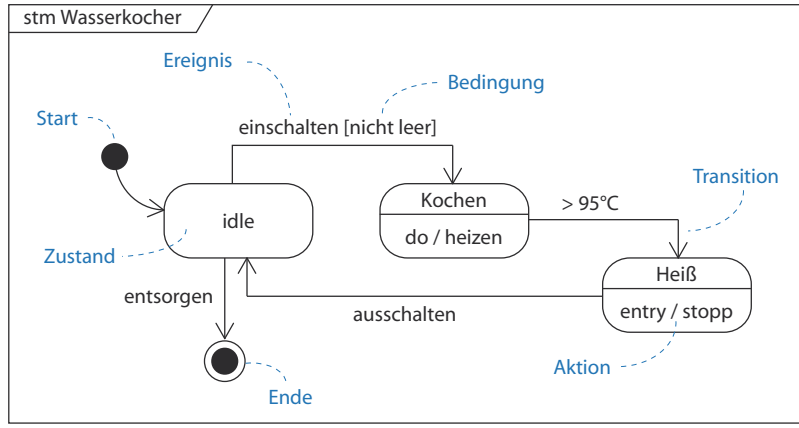


Abbildung 3.12 Zustände eines Wasserkochers

Endlicher Automat

Zustandsdiagramme beschreiben einen endlichen Automaten. Ein solcher Automat hat eine abzählbare und somit endliche Menge an Zuständen, die er annehmen kann. Bis das System untergeht, indem die entsprechende Komponente gelöscht wird, verbleibt der Automat in einem dieser Zustände. Ein Zustand verkörpert eine im System enthaltene Information, wiedergegeben durch ein oder mehrere Attribute. Die Pfeile zwischen den Zuständen im Diagramm geben die möglichen Übergänge an und durch welche äußeren Ereignisse ein solcher Zustandswechsel ausgelöst wird. Bei jedem Eintritt, Austritt und manchmal auch beim Verbleib in einem bestimmten Zustand kann die Ausführung einer Aktion definiert werden. Bei einem Wasserkocher wäre dies beispielsweise das Ausschalten der Heizspirale, wenn das Wasser seine gewünschte Temperatur erreicht hat und das System in den Zustand „Wasser heiß“ übergeht. Zuvor sorgt die Heizspirale für eine kontinuierliche Heizleistung. Zustandsdiagramme wie die der Abbildung 3.12 bestehen aus Zuständen, Transitionen und Ereignissen, die wir im Folgenden etwas eingehender besprechen wollen.

Zustand

Zustände sind vorübergehend stabile Stadien

Zustände sind Attraktoren, das heißt, sie sind im Zeitverlauf eines dynamischen Systems invariante, sich nicht ändernde Phasen. Oder etwas einfacher ausgedrückt: In einem Zustand befindet sich ein labiles System momentan in einem stabilen Stadium. Erst ein entsprechendes Ereignis, für das dieser Zustand empfänglich ist, vermag den temporär stabilen Zustand aus dem Gleichgewicht zu bringen und in einen anderen Zustand überzuführen. Wie lange ein System in einem bestimmten Stadium verweilt, ist nicht vorhersehbar. Andernfalls handelt es sich um eine Aktivität, die eine bestimmte Zeit in Anspruch nimmt und nicht – theoretisch – unendlich lange andauern kann. Diese Unterscheidung ist wichtig, da fälschlicherweise vielfach Aktivitätsdiagramme anstelle von Zustandsdiagrammen verwendet werden. Ein Zustand kann zwei Dinge tun: auf ein Ereignis warten oder eine Aktion ausführen.

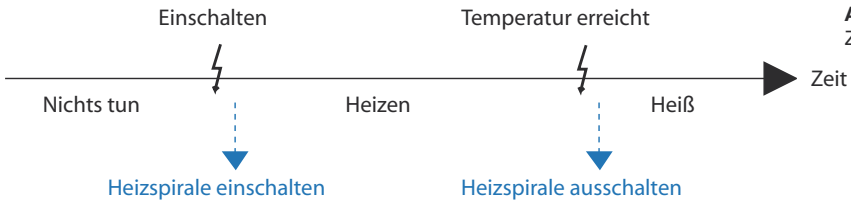


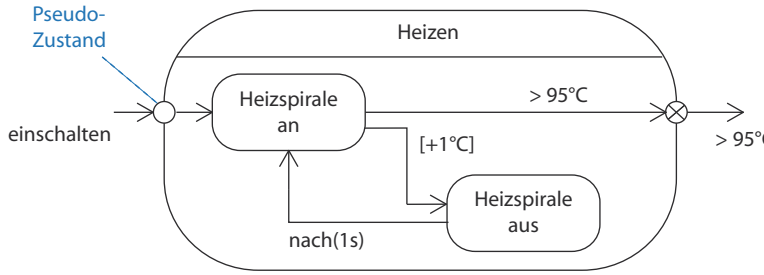
Abbildung 3.13
Zustandsübergänge

Unterzustand

In einem Zustand verändern sich die für die jeweilige Betrachtung relevanten Attribute des Systems oder Komponenten nicht oder nicht bemerkbar. Sie bleiben stabil. So ist im Zustand „Heizen“ die aktuelle Temperatur aus Sicht dieses Modells unerheblich. Erst die Erfüllung der Bedingung, eine bestimmte Temperatur zu erreichen, erzwingt einen Zustandswechsel. Blicken wir hingegen tiefer in den Zustand des Heizens, offenbart sich, dass sich bei diesem Produkt die Heizspirale in Abhängigkeit der Temperatur ein- und ausschaltet, um eine Überhitzung zu vermeiden, wie es die Abbildung 3.13 illustriert.

Mit diesen als Unterzustände oder auch als Kompositionszustände bezeichneten Diagrammen lassen sich Zustände weiter dekomponieren und damit hierarchisch in immer detailliertere Betrachtungen zerlegen. Wie bei den Aktivitätsdiagrammen kann die Dekomposition im selben Diagramm oder außerhalb, in einem eigenen Diagramm, erfolgen. Mithilfe von sogenannten Pseudo-Zuständen werden die Ein- und Ausgänge des übergeordneten Zustands übernommen. Manche Notationsformen verzichten auf die Darstellung der Pseudo-Zustände und verlinken die Transition direkt mit dem Zustand.

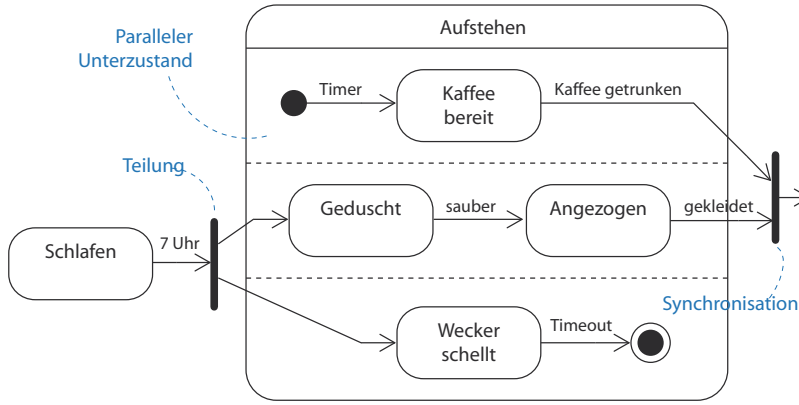
Abbildung 3.14
Unterzustand



Parallele Abläufe

Ebenfalls vergleichbar mit den Aktivitätsdiagrammen lassen sich in einem Unterzustand gleichzeitig stattfindende Abläufe modellieren. Hierbei wird der Zustand durch eine gestrichelte Linie in – als Submaschinen bezeichnete – parallele Ausführungsstränge aufgeteilt. So verdeutlicht das Beispiel der Abbildung 3.15, dass dem Schlaf drei parallele Abläufe folgen. Ausgelöst durch die morgendliche Stunde klingelt der Wecker, der Kaffee beginnt zu kochen und wir kleiden uns an für den bevorstehenden Tag. Angekleidet und mit Koffein gestärkt endet der Zustand Aufstehen und mündet – jedenfalls bei mir – im morgendlichen Stau oder, wenn das Auto streikt, in der Schlange an der Bushaltestelle.

Abbildung 3.15
Parallele Zustände



Transition

Transitionen spezifizieren Zustandsübergänge

Eine Transition zeigt die Richtung der Zustandsänderung an. Dabei signalisiert der Pfeil, auf welchen Zustand ein bestimmtes Ereignis einwirkt und welcher Zustand der Transition folgt. Die Bezeichnung einer Transition besteht aus der Auflistung der Ereignisse, optionalen Bedingungen und der damit möglicherweise verbundenen Ausführung von Aktionen. Es bestehen zwei Möglichkeiten, die bei einem Zustandswechsel auszuführende Aktion anzugeben: entweder als Aktion in der Transition selbst oder als Eintritts- oder Austrittsoperation im jeweiligen Zustand. Die Syntax für die Beschriftung der Transition ist:

```
Ereignis1, Ereignis2, .../[Bedingung]/Aktion1, Aktion2, ...
```

Die Bedingung wird wie bei einem Aktivitätsdiagramm als boolescher Ausdruck in eckigen Klammern formuliert. Er muss wahr sein, bevor der Zustandswechsel, ausgelöst durch ein Ereignis, erfolgen kann. Transitionen können sich wie beim Aktivitätsdiagramm teilen und synchronisieren, symbolisiert durch einen Balken.

Ereignis

Ein Ereignis ist ein relevanter Vorfall

Ein Ereignis ist ein markanter Vorfall, welcher unsere Aufmerksamkeit erlangt und zu einer Veränderung des wahrnehmbaren Zustands führt. Ereignisse in der technischen Welt vermögen die momentane Ruhe eines Zustands zu durchbrechen und das System zu einer Handlung veranlassen. Dies mündet in einem neuen Zustand. Ereignisse sind in der Regel an eine Transition geknüpft. Diese zeigen den Pfad einer Ereigniskette auf, also wo ein Ereignis auftritt und zu welchem Zustandswechsel das System veranlasst wird. Je nach der auslösenden Quelle wird zwischen folgenden Ereignissen unterschieden: Aufruf, Signal, Veränderung und Zeit.

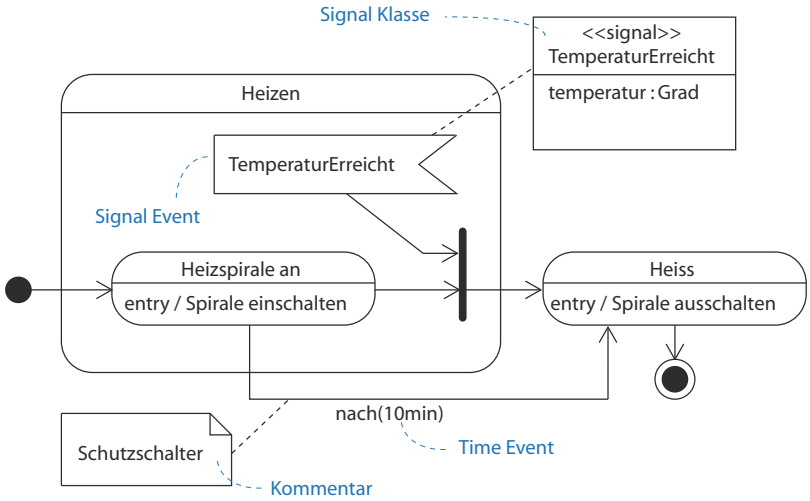


Abbildung 3.16
Signal-Event

- *Aufruf.* Ein aufrufendes Ereignis löst eine Operation auf einem Zustand aus, hat aber selbst keine weiteren Informationen. Diese Operation steht für einen Methodenaufruf einer Klasseninstanz (Objekt). Der Zustandswechsel wird dabei durch das Beenden einer zuvor ausgeführten Aktion oder durch die Erfüllung einer Bedingung ausgelöst. Ein durch Aufrufe getriebenes Zustandsdiagramm ist im Grunde ein Aktivitätsablauf und weniger die Darstellung eines endlichen Automaten. Der Zustand entspricht dabei eher einer Aktivität, welche nach dem Beenden der Operation verlassen wird.
- *Signal.* Es ist eine asynchrone Ein-Weg-Übertragung von Informationen von einem Objekt zu einem anderen. Signale stellen das dar, was wir im Allgemeinen unter Ereignissen verstehen. So ist das Erreichen einer bestimmten Temperatur im Wasserkocher oder ein Mausklick beispielhaft für solche Ereignisse, deren Auftreten zu einem beliebigen, nicht vorhersehbaren Zeitpunkt erfolgt. Signale werden als Klassen ohne Operationen und mit dem Stereotyp *Signal* modelliert.
- *Veränderung.* Ein Veränderungsereignis tritt auf, wenn eine bestimmte Bedingung erfüllt ist. Statt das Erreichen der gewünschten Wassertemperatur durch ein Thermostat zu signalisieren, ließe sich dieses Ereignis auch als Bedingung für das Erreichen einer bestimmten Temperatur darstellen. Veränderungen werden als boolescher Ausdruck in der Form, `wenn(Bedingung war)`, formuliert.
- *Zeit.* Zeitereignisse werden beim Erreichen eines bestimmten absoluten Zeitpunkts oder nach Ablauf einer definierten Zeitspanne ausgelöst. Sie werden als `wenn(Datum ist gleich)` oder `nach(Zeitspanne)` formuliert. Dabei ist die Zeiteinheit unbedingt anzugeben. Es macht einen Unterschied, ob eine Sekunde oder ein Monat gemeint ist.

Abbildung 3.16 illustriert die explizite Darstellung eines Signalereignisses. Ein Objekt der Klasse „Thermostat“ sendet ein entsprechendes Signal an die Heizeinheit des Wasserkochers. Mit dem eingebuchteten Fähnchen wird die Verbindung zu einem Objekt außerhalb angedeutet.

3.3.3 Use-Case-Diagramm

Use-Cases beschreiben die fachliche Sicht

Syntaktisch sind Use-Cases denkbar einfach. Sie bestehen im Grunde aus einem Oval und einer dahinter stehenden textuellen Beschreibung des Ablaufs sowie als Strichmännchen symbolisierten Akteuren. Trotzdem wurden über fast kein anderes Thema von UML so viele Bücher geschrieben. Um Use-Cases zu verstehen, müssen wir zurück auf deren Entstehung blicken. UML als grafische Notation entstand aus dem Bedürfnis, die Struktur und das Verhalten von Software als exaktes Modell zu beschreiben. Dieser eher technische Ansatz fokussiert sich auf das Zusammenspiel der einzelnen Bausteine, in der Regel Klassen, zu einem System und spricht damit die an der Implementierung beteiligten Personen an. Ivar Jacobson (Jacobson, 2000) bereicherte diesen strukturellen Ansatz des Softwareentwurfs um eine gemeinsame Sprache mit Fachexperten, die Use-Cases. Nicht mehr der

Aufbau eines Systems stand im Fokus, sondern dessen Funktionalität. Use-Cases konzentrieren sich auf die fachliche Tätigkeit und schildern die Nutzung des Systems aus Sicht der Anwender. Use-Cases und Klassen – damit sind Komponenten ebenso gemeint – sind zwei völlig verschiedene Dinge. Use-Cases fragen danach, *was* ich mit einem System machen kann, die Klassen fragen, *wie* das System zu bauen ist. Unglücklicherweise wurde beides zu einer gemeinsamen Sprache zusammengefasst und behauptet, die Use-Cases würden alle anderen Sichten verbinden.

Use-Cases spezifizieren das Verhalten eines Systems ausgelöst durch eine externe Instanz zur Erfüllung eines bestimmten Bedürfnisses. Eine externe Instanz wird dabei als Akteur bezeichnet. Dahinter verbirgt sich eine Rolle, die ein bestimmtes Verhalten und Interessen zusammenfasst. Wir alle nehmen im Leben unterschiedliche Rollen ein: als Vater, als Bankkunde, Bahnreisender, Manager oder Buchhalter. Jeder dieser Akteure oder Rollen hat unterschiedliche Anforderungen an ein System. Der Use-Case beschreibt einen Dialog zwischen einem solchen Akteur und dem System, eine in sich geschlossene Aufgabe zu erfüllen. Dabei steht das Bedürfnis im Vordergrund, welches ein Akteur mithilfe des Systems befriedigen möchte. Der Dialog entspricht dem Wechselspiel des Tischtennis. Auf eine Aktion des Akteurs antwortet das System und gibt die Kontrolle zurück an den Akteur. Ein solcher Dialog wird immer durch einen Akteur ausgelöst. Use-Cases sind aber keine exakten Modelle im Sinne von Klassendiagrammen. Es ist ein Hilfsmittel, um die Tätigkeiten der späteren Anwender zu verstehen und die einzelnen Schritte mit dem System in Bezug zu setzen. Wir wollen verstehen, wie eine Aufgabe vonstatten geht und welche Regeln, Alternativen und Ausnahmebedingungen dabei gelten. Im Grunde entsprechen Use-Cases der Beschreibung einer Bedienungsanleitung. Wir erfahren, wie der Benutzer mit dem System interagiert, jedoch nicht, wie das System diese Aufgabe bewältigt und welche Komponenten dabei involviert sind.

Use-Cases erzählen einen Dialog zwischen Anwender und System

Damit unterscheiden sich Use-Cases von der klassischen Anforderungsanalyse. Ein System wird nicht durch die Aufzählung bestimmter, zu erfüllender, Merkmale beschrieben, sondern durch das dynamische Verhalten. Die Interaktion zwischen Akteur und System wird als sequenzieller Ablauf modelliert. Use-Cases setzen die einzelnen Anforderungen in einen Zusammenhang. Statt nach den Features zu fragen, wird ergründet, was der Anwender mit Hilfe des Systems tun möchte und wie er dabei mit dem System interagiert. Deshalb eignen sich Use-Cases als effektives Werkzeug, innerhalb von Workshops die Bedürfnisse mit den unterschiedlichsten Interessengruppen zu ermitteln, ohne über technisches Wissen und Modellierungskompetenz verfügen zu müssen.

Persona und Szenarien

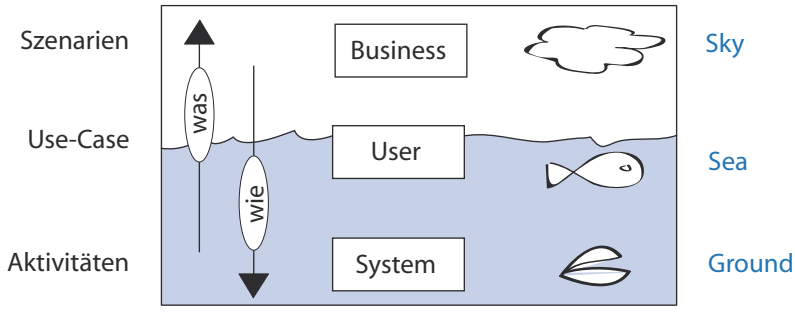
So weit die Theorie. In der Praxis tendieren Use-Cases dazu, wie für Klassen und Komponenten einen exakten, mit allen Eventualitäten versehenen Ablauf zu definieren. Das eigentliche Ziel, das unbekannte Wesen außerhalb des Systems zu verstehen, verliert sich in einer technisch gefärbten Ablaufbeschreibung. Die Essenz des Use-Case geht zwischen unzähligen Alternativabläufen und der Behandlung von Ausnahmebedingungen unter. Die

Reduktion des individuellen Benutzers auf eine bestimmte Rolle in Form des Akteurs ignoriert zudem die individuellen Bedürfnisse und Handicaps verschiedener Personen. Unbestritten, Use-Cases sind eine gute Sache, doch sollten wir diese eher im Sinn von Persona und Szenarien verwenden. *Persona* beschreiben eine bestimmte Ausprägung oder einen Personenkreis. Anstelle des anonymen Anwenders sprechen wir vom jugendlichen Technologiefreak mit dem modernsten Handy oder einer aufgeschlossenen älteren Dame, wenn wir analysieren wollen, wie bestimmte Personen mit dem System interagieren. Szenarien verzichten darauf, alle Eventualitäten in einen Use-Case zu packen. Stattdessen wird ein konkreter Fall mit einem gradlinigen Ablauf aufgezeichnet. Ob das Handy bereits für den Ticketkauf registriert ist oder nicht, sind zwei verschiedene Szenarien. Szenarien tendieren weniger stark dazu, bereits tief in den Lösungsentwurf absteigen zu wollen.

Use-Case-Level

Doch vorerst zurück zu den Use-Cases und deren Bezug zum Architekturentwurf. So wie Alistair Cockburn (Cockburn, 2001) bildlich darstellt, ist zwischen verschiedenen Use-Case-Level zu unterscheiden. Wie in Abbildung 3.17 schematisch erläutert, können wir zwischen verschiedenen Abstraktionsebenen unterscheiden: den Wolken, der Wasseroberfläche und dem Meeresgrund. Use-Cases in den Wolken beschreiben die Anforderungen aus Sicht des Business. Hierzu eignen sich besonders die zuvor skizzierten Szenarien. Diese Ebene sorgt für ein allgemeines Verständnis zwischen Geschäft und IT. Auf der See-Ebene werden die Anforderungen an das System aus dem Blickwinkel späterer Nutzer spezifiziert. Hier werden die Geschäftsregeln und Abläufe zu ersten Lösungsansätzen in Bezug gebracht. Auf dieser Stufe arbeiten Business-Analysten und Architekten eng zusammen, um die Geschäftsanforderungen zu einer tragfähigen Lösung zu entwickeln. Auf dem Meeresgrund werden die Abläufe innerhalb des Systems aufgezeigt und mit den zuvor besprochenen Aktivitätsdiagrammen modelliert. Die Zeichnung illustriert, dass Use-Cases auf verschiedenen Abstraktionsebenen und unterschiedlichem Detaillierungsgrad existieren. Dabei werden die Use-Cases der Geschäftsebene schrittweise in konkrete Systemanwendungsfälle übersetzt.

Abbildung 3.17
Use-Case-Level



Klassische und romantische Sicht

Bevor wir uns der etwas detaillierteren Besprechung der Use-Cases zuwenden, sei der Vergleich von Daryl Kulak (Kulak & Guiney, 2000) erwähnt, um den Unterschied zwischen der Beschreibung eines Systems durch Use-Cases

und Klassenmodell zu illustrieren. Dazu soll uns das Bild zweier Personen dienen, die ein Motorrad in unterschiedlicher Weise sehen. Der Fahrer erzählt aus der romantischen Sicht, was man mit einem Motorrad machen kann, und von dem damit verbundenen Fahrgefühl. Der Mechaniker sieht dasselbe Ding aus der klassischen Sicht. Er beschreibt, aus welchen Einzelteilen das Motorrad besteht und wie diese zusammen funktionieren. Use-Cases sehen im übertragenen Sinn ein System aus der romantischen Sicht und konzentrieren sich darauf, zu erläutern, was der Benutzer damit machen kann.

Use-Cases bestehen aus Akteuren, Use-Case-Blasen und deren dahinterliegenden Spezifikationen. Das Use-Case-Diagramm fasst mehrere Use-Cases grafisch zusammen und ordnet diese gegebenenfalls einem System oder Subsystem zu. Use-Case-Diagramme wie die der Abbildung 3.18 stellen die einzelnen Use-Cases und Akteure zudem in einen Gesamtzusammenhang. Quasi im Schnelldurchgang wenden wir uns nun den einzelnen Elementen zu.

Akteur

Akteur verkörpert eine Rolle

Wie bereits eingangs erwähnt, steht ein Akteur für eine bestimmte Rolle. So wie Klassen ein allgemeines Verhalten aller davon abgeleiteten Objekte definieren, legen Rollen stereotypisches Handeln fest. Akteure können Computer, Menschen und andere Applikationen darstellen, die in irgendeiner Form mit dem System interagieren. Ein Use-Case wird immer über den sogenannten primären Akteur initiiert. Auch bei Akteuren bestehen die Möglichkeiten einer Generalisierung. Hierbei erben Akteure das Verhalten der übergeordneten Klasse. Ob dies der Sache dienlich ist – ein für alle verständliches Modell zu erstellen –, sei dem Leser selbst überlassen.

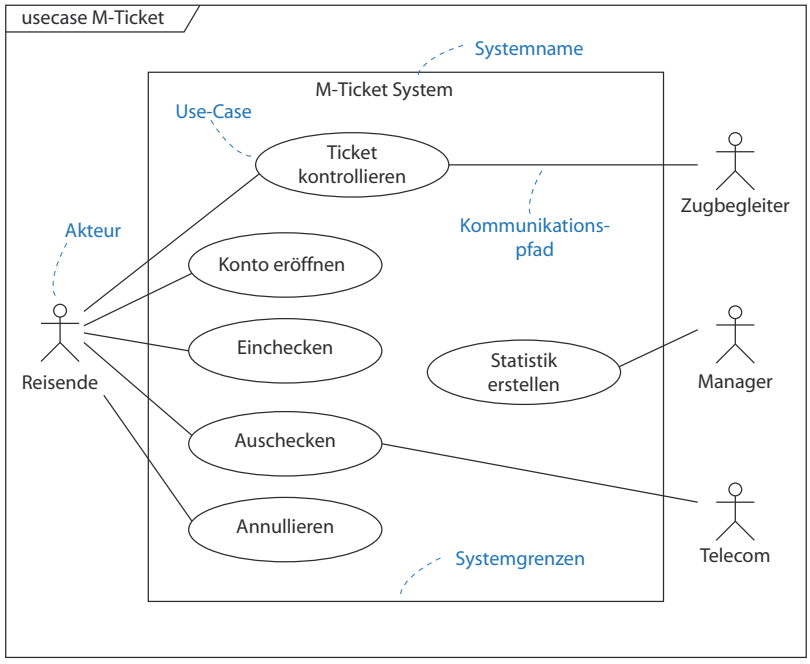


Abbildung 3.18
Use-Case-Diagramm

Use-Case

Ein Use-Case repräsentiert eine Funktion aus Sicht des Akteurs

Ein Use-Case ist etwas, das der Akteur mithilfe des Systems tun will. Er fasst einen Ablauf oder Dialog zusammen. Was das System ist, hängt einmal mehr von der jeweiligen Betrachtungsebene ab. Auf Geschäftsebene stellt das System eine Organisation dar und die Use-Cases sind Geschäftsprozesse. Auf der Benutzerebene handelt es sich um die zu bauende Applikation oder den Applikationsverbund. Entsprechend verhält es sich mit den Akteuren. Auf Geschäftsebene sind es die Kunden der Organisation und auf Benutzerebene die direkten Nutzer, die das System bedienen werden. In einem Use-Case-Diagramm werden diese Systemgrenzen und die Akteure sowie durch sie initiierte Use-Cases bildlich dargestellt. Ein Use-Case-Diagramm ist an und für sich nur eine visuelle Aufzählung der Anwendungsfälle, die für sich alleine stehen. Erliegen Sie nicht der Versuchung, funktionale oder sonstige Abhängigkeiten in Use-Case-Diagrammen abzubilden. Jeder Use-Case ist im Prinzip eine isolierte Betrachtung einer bestimmten Tätigkeit zusammen mit dem betrachteten System. Möglichkeiten, Teile von einzelnen Use-Cases durch Vererbung, Inkludieren und Erweiterung für andere wiederverwendbar zu machen, sollten nur sehr sparsam angewandt werden. Sie entstammen dem Denken in Funktionen und weniger dem Blick über die Schultern des Anwenders. Solche Diagramme sind zudem schwer verständlich, gerade für Personen, denen das objektorientierte Konzept wenig vertraut ist. In einigen Darstellungen wird der Kommunikationspfad mit einem Pfeil Richtung Use-Case untermalt. Dies ist nicht notwendig, da der Use-Case immer durch den Akteur angestoßen wird.

Spezifikation

Spezifikation beschreibt den Dialog in Einzelschritten

Die eigentliche Arbeit und damit der Mehrwert von Use-Cases liegen in deren Spezifikation. Diese meist textuell abgefassten Beschreibungen definieren die Vor- und Nachbedingungen und illustrieren den Dialog zwischen Akteur und System als sequenziellen Ablauf. Ein solcher Ablauf ist mit einem fließenden Gewässer vergleichbar, mit einer vorgegebenen Richtung und verschiedenen Verzweigungen, die wieder zur Hauptader zurückkehren können. Verzweigungen können einfache Bedingungen oder Schleifen sein. Diese Elemente sind jedoch sparsam zu verwenden, um nicht die Essenz des Use-Case unter einer Vielzahl von Abzweigungen unterschiedlichster Art zu verlieren. Auf die Gefahr hin, mich zu wiederholen: Use-Cases sind kein Ersatz für das Softwaredesign. Statt ähnlich dem Programmcode die korrekte Eingabe in einer Schleife mit anschließender Prüfung zu schreiben, ließe sich dieselbe Aussage mit einem Satz formulieren: Das System fordert eine gültige Eingabe. Es ist schließlich die spätere Aufgabe des Architekten oder Softwaredesigners, dem „Was“ des Business-Analysten ein „Wie“ in Form einer Lösungsskizze nachzuliefern. Wird beispielsweise bereits im Use-Case die Prüfung einer Kreditkarte mit der Eingabe eines Pin-Codes festgelegt, so werden damit alle biometrischen Möglichkeiten ausgeschlossen, die Authentifizierung des Bankkunden zu lösen. Dies ist grundsätzlich nicht falsch, wenn die Anforderung oder bestehende Systeme eine solche Lösung bereits

vorschreiben. Andernfalls sollte sich der Use-Case auf die Dokumentation des damit beabsichtigten Zwecks beschränken.

Use-Case Name (und ID)	Name: **Einchecken**
Kurzbeschreibung (ein Satz)	Kurzbeschreibung: Starten einer neuen Reise am Touch-Point
Auslösender Akteur	Primärer Akteur: Reisender
Beteiligte Akteure	Sekundäre Akteure
Müssen erfüllt sein, damit Use-Case starten kann	Vorbedingung: Reisender hat gültiges Konto
Schritte des Hauptablaufs	Basisablauf: 1. Kunde bewegt Handy an den Touch-Point 2. System signalisiert dem Handy die Erkennung 3. Handy fordert Reisenden auf, Check-In zu bestätigen 4. System eröffnete Reise 5. Handy zeigt erfolgreiche Eröffnung an
Systemzustand nach Ausführung	Nachbedingung: Reise ist eröffnet
Abweichende Abläufe	Alternativabläufe: Wenn Kunde in 3 ablehnt, bricht der Dialog ab

Abbildung 3.19
Use-Case-Spezifikation

Spezifikation nichtfunktionaler Anforderungen

Use-Cases sind eine gute Sache, um funktionale Anforderungen zu detaillieren. Es ist aber auch so, dass Use-Cases nicht dazu geeignet sind, alle Anforderungen zu spezifizieren. Insbesondere nichtfunktionale Anforderungen, die Qualitätsattribute und Rahmenbedingungen, bedienen sich nach wie vor der klassischen Spezifikation von Systemmerkmalen in Form von „Das System soll". UML kennt im Gegensatz zu SysML noch kein Diagramm zur Darstellung und Verknüpfung von solchen Anforderungen. Es ist zu erwarten, dass auch UML demnächst dahingehend erweitert wird. Eine Abhilfe ist beispielsweise, zusätzliche Anforderungen, die einen Use-Case betreffen, durch eine stereotypisierte Notiz oder eine Instanz einer Anforderungsklasse einem Anwendungsfall zuzuordnen.

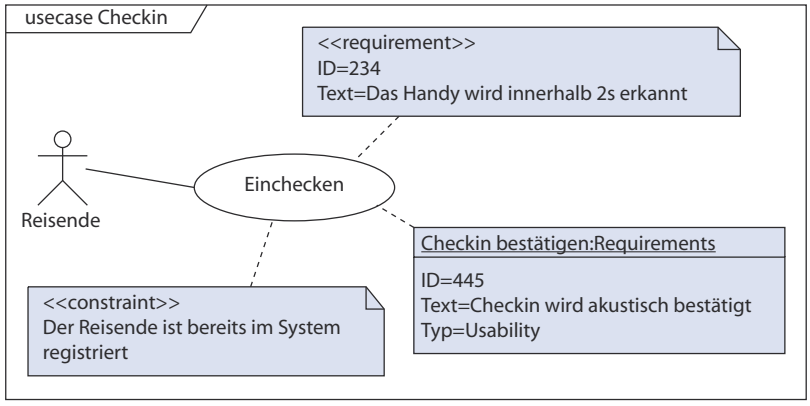


Abbildung 3.20
Use-Case um Anforderungen ergänzen

3.4 Interaktion

Interaktion zeigt den Meldungsverlauf zwischen Komponenten

Damit sind wir bei den Interaktionsdiagrammen angekommen, also bei jenen Diagrammen, die das Zusammenspiel mehrerer Bausteine über den Meldungsverlauf illustrieren. Das Sequenz- und Kommunikationsdiagramm ermöglichen, wenn auch auf unterschiedliche Weise, die Darstellung eines solchen Interaktionspfads. Diese Diagramme sind jedoch zu aufwändig, um für die Modellierung einer Klasse oder einer einzelnen Operation eingesetzt zu werden. Hier eignen sich, wenn deren Visualisierung notwendig ist, die zuvor besprochenen Aktivitätsdiagramme wesentlich besser. Sequenz- und Kommunikationsdiagramme sind dazu gedacht, ein Szenario über verschiedene Komponenten und Subsysteme zu beschreiben. Es ist die Choreografie von Diensten zur Illustration eines bestimmten Ablaufs auf System- oder Subsystemebene. Es ist das in Szene Setzen eines Systemverhaltens auf einen äußeren Stimulus. Deshalb bieten sich Use-Cases als Ausgangspunkt für die Entwicklung solcher Interaktionsbeschreibungen geradezu an. Use-Cases dokumentieren als Sequenz von Einzelschritten die Interaktion mit dem System. Statt mit dem System als Ganzes wird nun die Kommunikation mit den jeweiligen Subsystemen oder Komponenten innerhalb des Systems in den Vordergrund gestellt.

3.4.1 Sequenzdiagramme

Sequenzdiagramme zeigen die an einer Interaktion beteiligten Teilnehmer und die Sequenz des Meldungsverlaufs. Teilnehmer sind dabei aktive Elemente wie Objekte oder Instanzen von Komponenten, aber auch Akteure. Neben diesen Objekten besteht ein Sequenzdiagramm aus Lebenslinien und an diesen festgemachten Meldungen sowie als Fragmente bezeichneten Erweiterungen. Lebenslinien sind gestrichelte, vertikale Linien unterhalb der Objekte, welche die Zeit zwischen der Instanzierung eines Objekts bis zu dessen Zerstörung anzeigen. Vielfach existiert kein explizites Kreieren und Löschen einer Instanz, da die Objekte die gezeigte Interaktion überdauern. Die Meldungen werden als horizontale Linie von den Lebenslinien des Senders zum Empfänger gezeigt. Die zeitliche Abfolge des Meldungsverlaufs ergibt sich durch die horizontale Anordnung. Die erste Meldung befindet sich zuoberst, die letzte zuunterst. Meldungen können synchron (volle Pfeilspitze), asynchron (offene Pfeilspitze) oder eine Antwort auf eine synchrone Meldung sein (gestrichelte Linie).

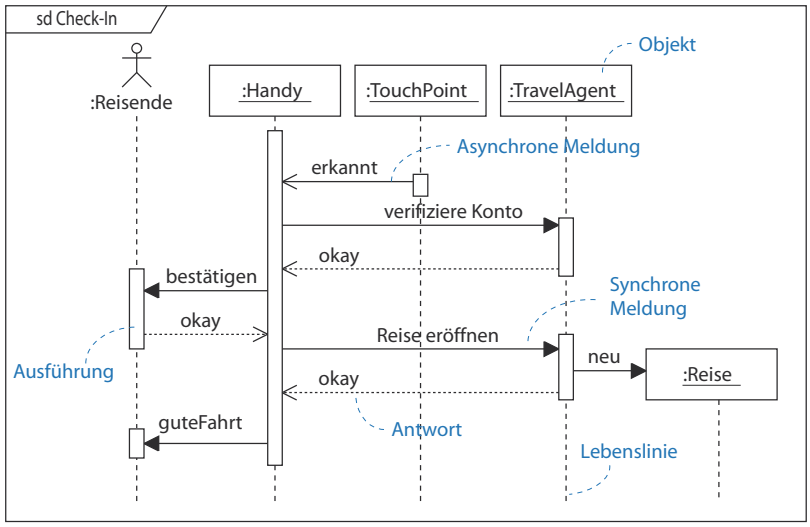


Abbildung 3.21
Sequenzdiagramm

Ausführungsbalken

Der vertikale Balken auf der Lebenslinie zeigt eine durchgehende Ausführung einer Funktion an. Der als Ausführungsspezifikation benannte Balken zeigt an, wo der momentane Fokus der Kontrolle liegt. Dies ist vergleichbar mit dem Aufrufstapel eines Programms. Alle darüber liegenden Aufrufe werden durch die darunterliegende Operation kontrolliert. Oder etwas bildlicher umschrieben: Wird die Funktion `zeichneBild` eines Objekts der Klasse `Bild` aufgerufen und diese ruft für alle seine Elemente deren Methode `zeichneElement` auf, so liegt der Fokus bis zum fertigen Bild bei `zeichneBild`. Oft werden die Ausführungsspezifikationen auch weggelassen, weil es in einem losen Zusammenspiel kein steuerndes Zentrum gibt oder die Darstellung keinen zusätzlichen Nutzen oder Information bringt. So wäre das Beispiel der Abbildung 3.22 auch ohne Ausführungsbalken, also den vertikalen Balken, eindeutig,

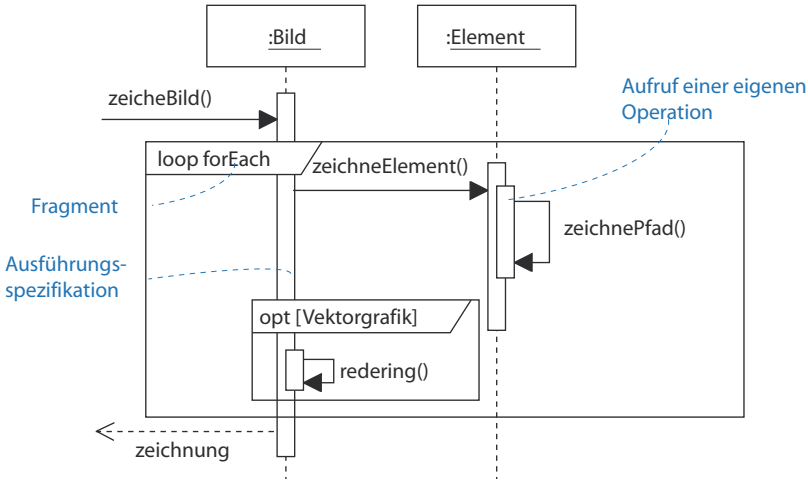


Abbildung 3.22
Ausführungsspezifikation

Fragmente

Wie das Beispiel zudem zeigt, ist es nun auch im Sequenzdiagramm möglich, bedingtes und repetitives Verhalten zu modellieren. Seit UML 2 können Sequenzdiagramme in Bereiche aufgetrennt werden, diese bezeichnet man als *kombinierte Fragmente*. Ein solches als Fragment bezeichnetes Element besteht aus einem Operator mit einer Bedingungsklausel und einem Rahmen, der die davon betroffenen Meldungen umschließt. Diese Klausel (oder auch Wärter) wird als boolescher Ausdruck in eckige Klammern gesetzt. Die Fragmente lassen sich in Bedingungen, Schlaufen, Mehrfachverzweigungen und Referenzen aufteilen. Solche kombinierten Fragmente eröffnen neue Optionen in der Verwendung der bis jetzt etwas starren Sequenzdiagramme. Besonders ist hier die Möglichkeit von Interaktionsblöcken hervorzuheben. Sie erlauben es, Teile eines Szenarios in ein separates Diagramm auszulagern. Damit vereinfacht sich erstens das ursprüngliche Diagramm und zweitens werden dadurch wiederverwendbare Interaktionen geschaffen. Entscheidend an dieser Erweiterung ist weniger die Wiederverwendung als die Möglichkeit, eine komplexe Darstellung in mehrere kleinere und damit weniger fehleranfällige Interaktionsblöcke zerlegen zu können. Andernfalls wird das Arbeiten mit Sequenzdiagrammen schnell zum Albtraum. Das Verschieben einer Meldung bringt oftmals das ganze Diagramm durcheinander, wenn sich darauf unzählige Elemente befinden. Beispielsweise lässt sich der folgende Baustein zur Überprüfung der Bonität in das übergeordnete Szenario der Registrierung eines neuen Abonnementen der Abbildung 3.24 einbinden. Dabei funktioniert der Sequenzbaustein wie eine aufrufbare Operation. Es lassen sich Argumente als Parameter übergeben und das Resultat als Ergebnis zurücksenden. Die Signatur des Bausteins wird in der Kopfzeile des Fragments angegeben.

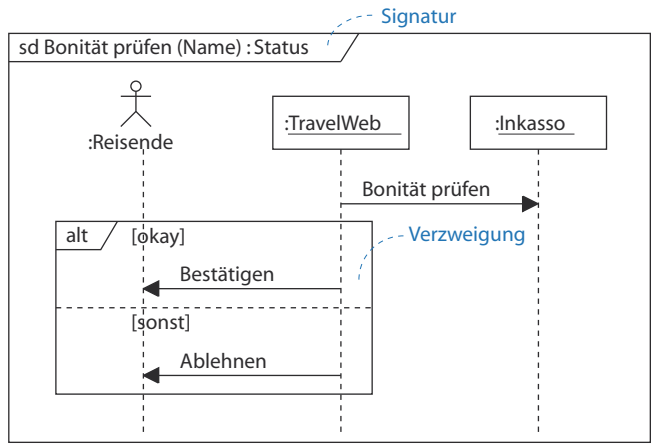


Abbildung 3.23
Definition eines Interaktionsbausteins

Tabelle 3.1 listet die wichtigsten Operatoren der kombinierten Fragmente auf. Je nach Typ hat ein Fragment ein oder mehrere Operanden, abgetrennt durch gestrichelte Linien. Ein Operand entspricht einer Ausführungssequenz. Die Bedeutung der Operanden hängt vom jeweiligen Operator ab. So sind es beim Operator *alt* Verzweigungen und bei *par* die gleichzeitige Ausführung

mehrere Programmfäden. Der Operator definiert, *wie* der Operand ausgeführt wird, und die Wächterbedingung sagt aus, *ob* er ausgeführt wird.

Tabelle 3.1
Fragment-Operatoren

Operator	Name	Beschreibung
opt	Optional	Optionale Ausführung, wenn die Bedingung wahr ist
alt	Alternative	Mehrfachverzweigung. Jener Operand wird ausgeführt, welcher wahr ist.
loop	Schlaufe	Wiederholung des Operanden solange die Bedingung wahr ist
ref	Referenz	Referenz auf eine andere Interaktion
par	Parallel	Parallele Ausführung der Operanden
critical	Exklusiv	Atomare Ausführung des Operanden ohne Unterbrechung

Zustände

Ebenfalls lassen sich Zustände und deren Übergänge im Sequenzdiagramm darstellen. Dabei wird der durch bestimmte Meldungen forcierte Zustandswechsel hervorgehoben. Auch hier gilt, dieses Konstruktionselement sparsam zu verwenden und nur für die jeweilige Betrachtung relevante Zustandsänderungen einzusetzen. Beispielsweise zeigt die Abbildung 3.24, dass das Konto erst verfügbar ist, wenn die Handy-Applikation `TravelApp` installiert und die Bestätigung an den Kunden versandt ist.

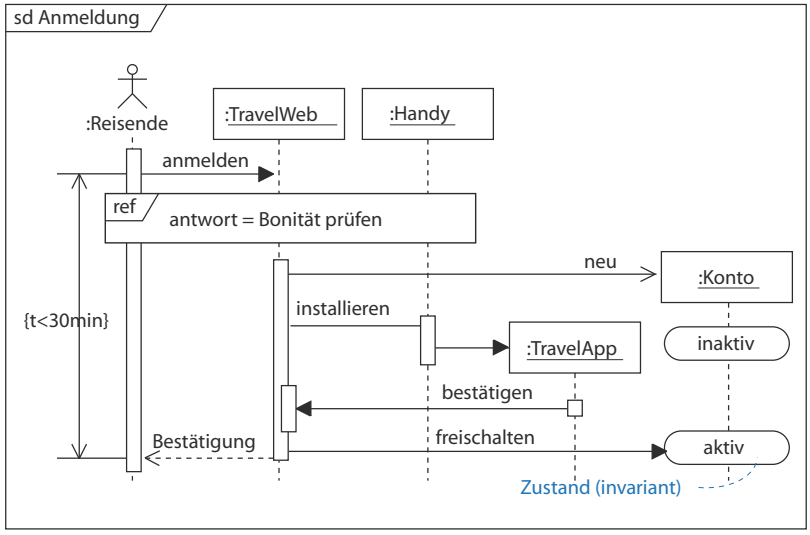


Abbildung 3.24
Referenz auf Interaktionsbaustein

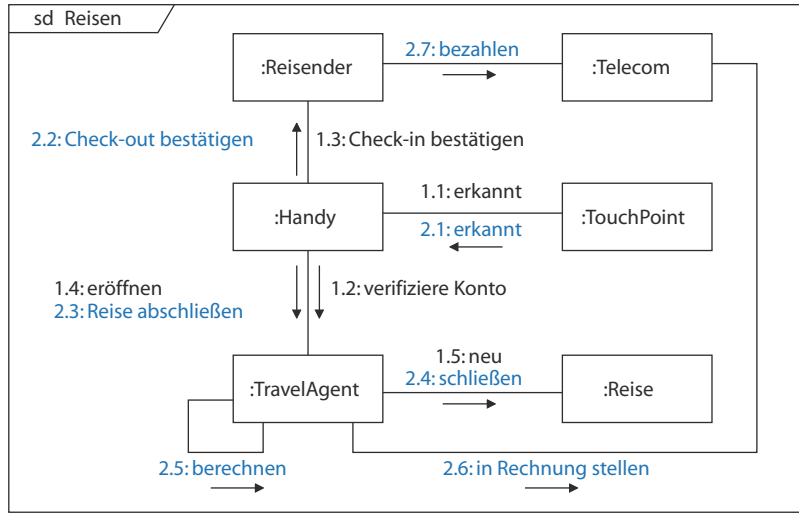
Wie bereits zu Beginn des Kapitels angedeutet, sind Sequenzdiagramme für technisch unversierte Personen nicht immer ganz einfach zu verstehen. Die Erweiterung durch verschiedenste Fragmente machen Sequenzdiagramme zu einem mächtigen Werkzeug in der Modellierung, erschweren jedoch deren Verständlichkeit. Der intensive Gebrauch von Fragmenten führt zu kaum

veränderbaren und außerdem zu schwer lesbaren Diagrammen. Manch verzweifelter Blick eines Modellierers geht auf den Versuch zurück, ein Fragment in einem überladenen Sequenzdiagramm verschieben zu wollen. Mein Rat: Verwenden Sie Sequenzdiagramme sparsam für die Darstellung wichtiger Szenarien und halten Sie sich an eine einfache Diagrammdarstellung. Eine Alternative zu den schwergewichtigen Sequenzdiagrammen stellen die im Folgenden aufgeführten Kommunikationsdiagramme dar.

3.4.2 Kommunikationsdiagramme

Kommunikationsdiagramme sind Objektdiagramme, in denen zusätzlich der Interaktionsverlauf durch nummerierte Meldungen eingezeichnet ist. Im Grunde sind die Kommunikationsdiagramme den zuvor beschriebenen Sequenzdiagrammen sehr ähnlich. Wurde in den Sequenzdiagrammen der zeitliche Verlauf über die horizontale Anordnung definiert, so sind es im Kommunikationsdiagramm Nummern und Pfeile, die den Meldungsfluss dokumentieren. Wie Abbildung 3.25 illustriert, sind mehrere, zeitlich voneinander getrennte Sequenzen in derselben Darstellung zu modellieren. Jede Sequenz wird in der Form *Sequenz.Schritt* nummeriert. So werden im gezeigten Beispiel im ersten Durchgang der Interaktionspfad des Einchecken und im zweiten der des Auscheckens abgebildet.

Abbildung 3.25
Kommunikationsdiagramm



Bedingungen und Schleifen lassen sich als Klausel zwischen Nummer und Meldungstext in eckige Klammern stellen, wie in der folgenden Darstellung angedeutet. Gerade hier offenbart sich der Vorteil gegenüber den Sequenzdiagrammen.

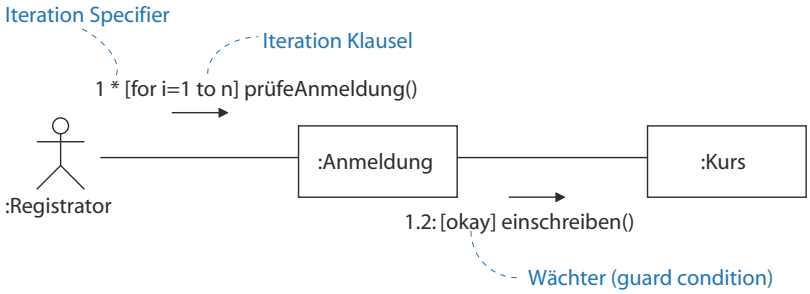


Abbildung 3.26
Bedingte Meldungen

Ob man für die Darstellung von Interaktionen das häufiger verwendete Sequenzdiagramm einsetzt oder dem Kommunikationsdiagramm den Vorzug gibt, ist im Prinzip Geschmackssache. Der Unterschied liegt vor allem in der Erweiterbarkeit der Diagramme und der Eignung für Handskizzen. Kommunikationsdiagramme lassen sich zu allen Seiten hin erweitern und das Hinzufügen neuer Interaktionspfade ist keine Hexerei. Zudem sind sie auch von Hand während eines Workshops leicht zu zeichnen. Kommunikationsdiagramme eignen sich hingegen kaum für die Darstellung umfangreicher Meldungsverläufe. Hier beweisen die Sequenzdiagramme ihren Vorteil eines klar geordneten, von oben nach unten zu lesenden Interaktionspfads.

3.4.3 Timing-Diagramme

Timing-Diagramme sind ebenfalls eine Erweiterung des UML-2-Sprachumfangs. Damit sollte die Lücke für die Modellierung von Echtzeitsystemen geschlossen werden. Timing-Diagramme sind sehr einfach und bestehen nur aus wenigen Elementen. Auf der Vertikalen werden die möglichen Zustände eines Objekts und auf der Horizontalen die fortschreitende Zeit dargestellt. Eine Treppenkurve zeigt an, nach welcher verstrichenen Zeit, welches Ereignis einen Zustandswechsel hervorruft.

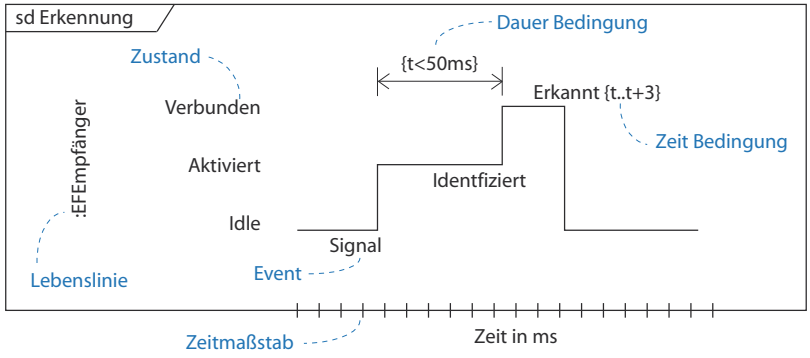


Abbildung 3.27
Timing-Diagramm

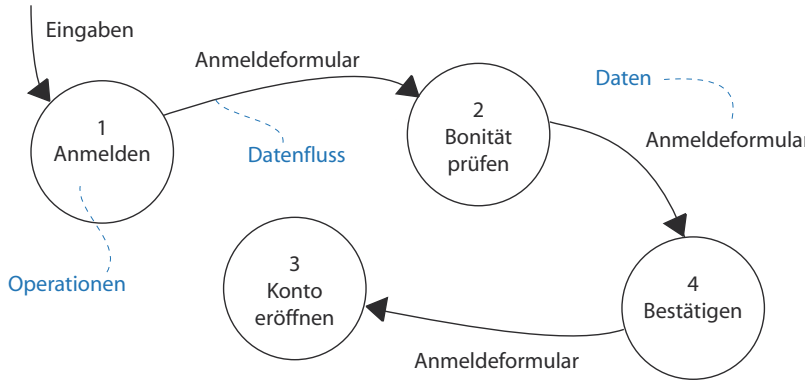
Das Diagramm wird von links nach rechts dem Zeitmaßstab folgend gelesen. Nach einer bestimmten durch eine Bedingung festlegbaren Maximaldauer hat die Komponente auf das Event oder den Stimulus zu reagieren. Als Antwort ändert die Komponente ihren Zustand. Mit dem Timing-Diagramm lassen sich, besser als im Zustandsdiagramm, getaktete Ereignis-Zustands-Wechsel illustrieren und dabei die maximale Dauer bis zu einer entsprechenden Reaktion darstellen. Indem mehrere Lebenslinien im Timing-Diagramm untereinandergelegt werden, ist die zeitliche Abhängigkeit zwischen mehreren Objekten sichtbar.

3.5 Objektorientierung

Die Ableitung, UML sei gleich Objektorientierung, gilt heute nicht mehr. UML ist eine Sprache mit einem breiten Einsatzspektrum. Trotzdem gehört zu UML ein Verständnis für objektorientierte Paradigmen. Im Schnelldurchgang werden wir nun die Grundkonzepte einer Welt aus Klassen und Objekten diskutieren. Dabei werden Begriffe der vorangehenden Einführung in UML nochmals aus dem Blickwinkel objektorientierter Sprachen reflektiert.

In den ursprünglichen – prozeduralen – Sprachen fand eine strikte Trennung zwischen Daten und den darauf angewendeten Operationen statt, wie die Abbildung 3.28 anhand eines Datenflussdiagramms illustriert. Die Verarbeitung glich einem Datenfluss, der an unzähligen Stellen vorbeiführte, die diese Daten bearbeiteten und sie dann weiterreichten. Im Zentrum standen die Daten und deren jeweiliger Informationsgehalt.

Abbildung 3.28
Datenfluss prozeduraler Sprachen



Objektorientierung ist eine neue Denkweise

Mit der wachsenden Größe der Programme und der damit verbundenen Aufteilung der Arbeiten auf verschiedene Entwicklungsgruppen waren die Daten zunehmend der limitierende Faktor. Eine weitgehend unabhängige Arbeitsweise war nur schwer möglich. Um dieser Komplexität zu begegnen, war eine höhere Abstraktion notwendig. Statt einer datengetriebenen, maschinenorientierten Programmierung sollte zukünftig in Verantwortlichkeiten

gedacht werden. Nicht mehr die Daten standen im Vordergrund, sondern ein Bündel von zusammengehörigen Operationen, die sich auf dieselben Daten bezogen. Daten und Operationen sollten zu eigenständigen logischen Blöcken verschmelzen: zu Klassen und Objekten. Es geht dabei nicht, wie oft falsch verstanden, nur um einen kontrollierten Zugriff auf die zuvor globalen Daten. Objektorientierung fordert eine neue Denkweise. Wir fragen nicht mehr, durch welche Attribute sich ein Ding beschreiben lässt, sondern danach, was es für uns tun kann. Als Vorbild diente die Natur, in der die Teilnehmer selbstständig handeln und doch ein funktionierendes Ökosystem bilden konnten.

Von daten- zu objektorientierten Programmen

Obwohl die Objektorientierung diesem Anspruch nach eigenständigen Mikrosystemen nicht vollständig zu genügen vermag, veränderte dieses Konzept die Art und Weise unseres Vorgehens. Die Informationstechnologie befreite sich von der datenverarbeitenden Sichtweise hin zu einer servicegetriebenen Architektur. In einem datenorientierten Ansatz besteht ein Programm aus einer Sammlung verschiedener Prozeduren oder Operationen, welche auf mehr oder weniger dieselben zentralen Daten einwirken. Im Prinzip kann jede Prozedur alle Daten in beliebiger Weise verändern oder löschen. Die Daten sind öffentlich. In einer objektorientierten Sprache besteht hingegen ein Programm aus zusammengehörigen Funktionsblöcken, in denen die gemeinsamen Daten privat sind. Ein Programm setzt sich nun aus einer Vielzahl kleiner Prögrämmchen zusammen. In der Praxis sind jedoch Klassen zu feingranular und hängen zu stark voneinander ab, um im Sinne der Metapher von lebenden Zellen weitgehend selbstständig zu agieren. An deren Stelle trat wenig später das Konzept der Komponenten. Die Objektorientierung ist im Grunde eher ein Programmierstil und weniger ein Architektur- oder Entwurfsmuster.

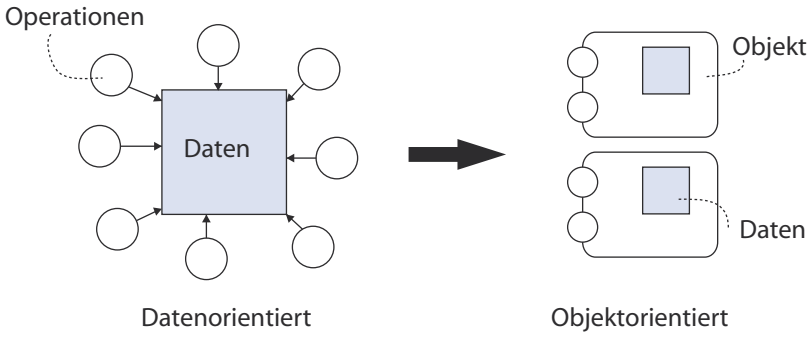


Abbildung 3.29
Von der Daten- zur Objektorientierung

3.5.1 Klassen und Objekte

Klassen sind Muster und Objekte sind konkrete Instanzen einer Klasse

Objektorientierung ist das Denken in Klassen und Objekten: Eine Klasse ist ein Modell oder ein Muster, ein Objekt stellt eine konkrete Instanz dieses Modells dar. Alles klar? Unsere Sprache macht es nicht einfach, die bei-

den Begriffe auseinanderzuhalten. Wir sprechen von Objekten, um abstrakte Gegenstände zu benennen, und verwenden Klasse in der Bedeutung einer Klassifizierung. In der objektorientierten Welt ist eine Klasse ein Muster oder die Beschreibung für ein gemeinsames Verhalten aller davon abgeleiteten Instanzen. Eine Klasse existiert nur in unserer mentalen Vorstellung. Eigentlich so, wie unser Erbgut einen Bauplan der Zelle enthält, anhand dessen der Organismus wieder eine neue Zelle erstellen kann. Klassen sind aber auch Module im Sinne der kleinsten Einheit einer Software in objektorientierten Programmen. Eigentlich interessiert in einer reinen objektorientierten Betrachtung die konkrete Implementierung dieser kleinsten Einheiten, den Klassen, nicht. Nur das sichtbare Verhalten und deren Eigenschaften sind für den Entwurf relevant. Dabei ist eine höhere Abstraktion als mit einem datenorientierten Ansatz möglich.

Metaklassen

Puristen mögen jetzt einwenden, dass Klassen als solche, im Sinne einer Instanz in einer Programmiersprache durchaus adressierbar sind. Konkret handelt es sich dabei aber nicht um Klassen, sondern um Objekte einer Metaklasse. Metaklassen sind Klassen von Klassen. Die im Programm adressierbare Klasse ist die Instanz der Metaklasse aller *Klassen*. In Analogie zum zuvor verwendeten Beispiel einer Zelle ist das Erbgut, welches den Bauplan einer Zelle enthält, selbst auch ein konkretes Objekt auf einer anderen Betrachtungsebene. Klar?

Jedes Objekt hat einen Zustand, ein Verhalten und eine eindeutige Identifikation

Jedes Objekt hat einen Zustand, ein Verhalten und eine eindeutige Identifikation. Der Zustand ist durch seine Attribute, sein Verhalten durch dessen Operationen und die Identifikation durch seine Adresse gegeben. Eine Adresse kann explizit durch einen eindeutigen Schlüssel oder implizit durch die Speicheradresse gegeben sein. Die Klasse definiert, welche Attribute ein Objekt aufweisen kann, und legt das grundsätzliche Verhalten durch die Implementierung der Operationen fest. Die Objekte dieser Klasse unterscheiden sich nur durch die jeweiligen Werte ihrer Attribute, die Operationen übernehmen die Objekte von ihrer Klasse.

3.5.2 Objektorientierte Konzepte

Das Objektmodell basiert auf den Konzepten der Kapslung, Vererbung und des Polymorphismus.

Kapslung

Kontrollierter Zugriff auf private Daten

Die Kapselung verbirgt die Daten und sorgt für einen kontrollierten Zugriff auf diese. Das Objekt kann von außen nicht in einen undefinierten Zustand versetzt werden; oder sollte es zumindest nicht. Die Kapslung verdeckt die konkrete Implementierung der Operationen und stellt über Schnittstellen ein definiertes Verhalten und letztendlich den Zugriff auf die inneren Daten bereit. Diese Schnittstellen stellen einen Kontrakt zwischen der Klasse und der Außenwelt dar.

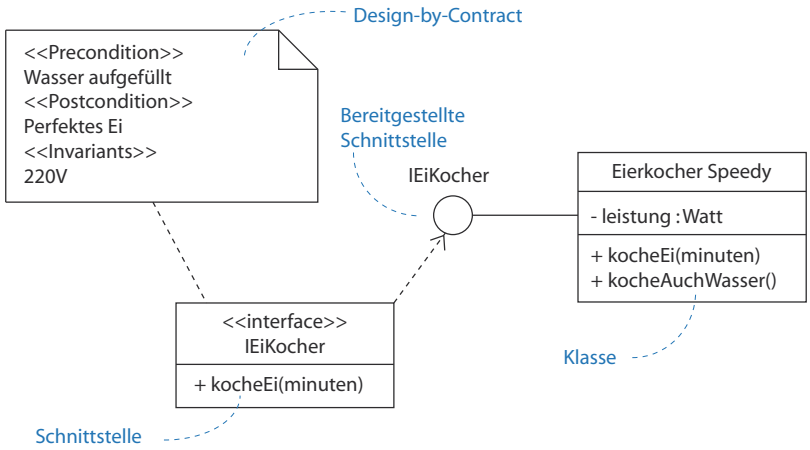


Abbildung 3.30
Schnittstellenspezifikation

Design by Contract

Bertrand Meyer hat unter dem Begriff *Design by Contract* oder kurz DBC definiert, dass eine Schnittstelle sich durch ihre Vor- und Nachbedingungen sowie die Invarianten exakt definieren lässt. Vorbedingungen sind Zustände und Voraussetzungen, die erfüllt sein müssen, damit eine Aktion durchführbar ist. Hierzu zählt auch die Gültigkeit der Übergabewerte. Nachbedingungen definieren den Zustand, in dem sich die Klasse nach der Ausführung der Operation befindet. Invarianten sind Aussagen, die unverändert über alle Instanzen und Methoden ihre Gültigkeit besitzen. Ursprünglich war der Begriff *Schnittstelle* die Spezifikation einer Operation im Sinne des von Bertrand Meyer postulierten Vertrags. Spätere Programmiersprachen wie Java führten eine explizite Unterscheidung zwischen der Definition einer Schnittstelle und deren Implementierung durch eine Klasse ein. Die Trennung zwischen Schnittstellenvertrag und deren konkreter Umsetzung ist die Basis für die zuvor angesprochene Serviceorientierung. Nicht mehr ein bestimmter Programmstil wie die Objektorientierung steht zur Diskussion, sondern die Bereitstellung von Leistungen über wohldefinierte Schnittstellen.

Nicht nur in der Programmierung ist diese Trennung eine empfehlenswerte Praxis. Auch die Spezifikation der Leistungsmerkmale von Geräten lässt sich mit Schnittstellen dokumentieren. In Abbildung 3.30 definiert beispielsweise der Interfacetyp `IEiKocher` den Service `kocheEi(minuten)`, ohne etwas über dessen konkrete Implementierung auszusagen. Der Schnittstellenvertrag definiert die Bedingungen, die für das Kochen eines Eis gelten, ohne eine bestimmte Implementierung zu nennen. Die Klasse `EierkocherSpeedy` ist dann eine mögliche Lösung.

Polymorphismus

Dynamische Bedingung

Polymorph bedeutet, dass eine Nachricht, welche an ein Objekt gesendet wird, unterschiedliche Operationen auslösen kann. Dieses Verhalten ist eng gekoppelt mit der Vererbung. Eine abstrakte Operation auf der Basisklasse wird in ihren abgeleiteten Klassen unterschiedlich implementiert. Wird die Operation der Basisklasse aufgerufen, so wird erst zur Laufzeit die tatsächlich aufzurufende Operation ermittelt und in der jeweiligen Subklasse aufgerufen. Wir sprechen dabei auch von einer dynamischen Bindung. Im Gegensatz zur statischen Bedingung in der Codierungsphase findet die Bindung erst zur Laufzeit statt. Im folgenden Beispiel wird in `drawAll` dynamisch in Abhängigkeit von der konkreten Instanz die jeweilige `draw`-Operation aufgerufen.

Listing 3.1 Polymorphismus

```
abstract class Shape {
    public abstract void Draw();
}

class Circle : Shape {
    override public void Draw() {
        // Zeichne ein Kreis
    }
}

class Rectangle : Shape {
    override public void Draw() {
        // Zeichne ein Rechteck
    }
}

//…
public void drawAll(Shape[] shapes) {
    foreach (Shape s in shapes) {
        s.Draw();
    }
}
```

Vererbung

Wiederverwendung von Gemeinsamkeiten

Hinter der Vererbung steht der Wunsch der Wiederverwendung. Gemeinsamkeiten sollen in einer übergeordneten Klasse implementiert und automatisch an die davon abgeleiteten Klassen vererbt werden. Damit sind Änderungen nur an einem einzigen Ort, in der Basisklasse, vorzunehmen und werden ebenfalls automatisch an die angeleiteten Klassen weitergegeben.

Wiederverwendung sollte nie treibender Faktor sein

Lange Zeit galt diese Vererbungsmöglichkeit als „Ei des Kolumbus“, um Redundanz zu vermeiden und die Wiederverwendung zu fördern. Dabei entstanden häufig degenerierte Vererbungsbäume, deren Basisklassen künstlich geschaffene Gemeinsamkeiten enthielten. Die impliziten, oft unerwünschten Seiteneffekte in den abgeleiteten Klassen durch Änderungen in der Basisklasse machten die Sache nicht besser. Die Verwendung einer abgeleiteten Klasse glich einem Eisberg. Nicht immer war klar, was unter der Oberfläche auf einen wartete bzw. mit welchem Preis von nicht gewollter Funktionalität die Wiederverwendung erkauft werden musste. Die Möglichkeit, die zuvor geforderte Kapslung für abgeleitete Klassen teilweise aufzuweichen, war ein

Freischein für manch sonderbare Konstruktion. Daten sollten, um das hier zu erwähnen, immer privat sein. Alles andere ist eine Verletzung des geforderten Geheimnisprinzips und erzeugt gewollt oder ungewollt eine hohe Kopplung zwischen den Klassen. Wie kann ich als Basisklasse die Erfüllung des Schnittstellenvertrags garantieren, wenn ich meinen Nachkommen den direkten Zugriff auf meine Daten erlaube?

Nur bei echter Vererbung anwenden

Die Vererbung ist aufgrund der aufgeführten Einschränkungen mit Bedacht einzusetzen und nur wirklich dort, wo eine Klasse eine Generalisierung im engeren Sinne darstellt. Eine solche liegt vor, wenn die abgeleitete Klasse im Grunde dieselben Fähigkeiten hat und diese durch ein zusätzliches Verhalten ergänzt werden. Beispielsweise zeigt Abbildung 3.31, dass die erweiterte Warteschlange die Fähigkeiten der Oberklasse nutzt und ihrerseits diese um zwei zusätzliche Operationen erweitert. In vielen Fällen, auch in der Literatur zitierte Beispiele, handelt es sich jedoch um abstrakte Klassen, welche die Implementierung einer Operation erzwingen. Hierdurch erhält die Oberklasse ein polymorphes Verhalten, was der eigentliche Zweck der missbräuchlichen Verwendung der Vererbung war.

Abbildung 3.31
Beispiel einer echten Vererbung

Interface statt Klassen

Stattdessen ist für viele Klassen ein interfaceorientierter Ansatz vorzuziehen. Hierbei wird die Implementierung einer Fähigkeit (Operation) von deren Anbieten über definierte Schnittstellen (Interfaces) getrennt. Damit können verschiedene Klassen dieselbe Fähigkeit implementieren, ohne eine Gemeinsamkeit im engeren Sinne der Vererbung aufweisen zu müssen. Bietet eine Klasse die Implementierung eines Interfaces an, kann dieser Service ohne Wissen über die jeweilige Klasse aufgerufen werden. Auch hier findet eine dynamische Bindung statt. Mit Interfaces wird man beispielsweise dem Umstand gerecht, dass, wie in der Abbildung 3.32 gezeigt, alle Vögel fliegen und auch „Nicht"'-Vögel sich in der Luft fortbewegen können. Eine gemeinsame Basisklasse zwischen Vögeln und Flugzeugen würde den Sinn einer Vererbung in der Tat arg strapazieren. Vielleicht ist der Vergleich etwas überzeichnet. Er sollte plakativ die Vorteile von Schnittstellen anstelle der propagierten Vererbungsmöglichkeiten objektorientierter Sprachen aufzeigen.

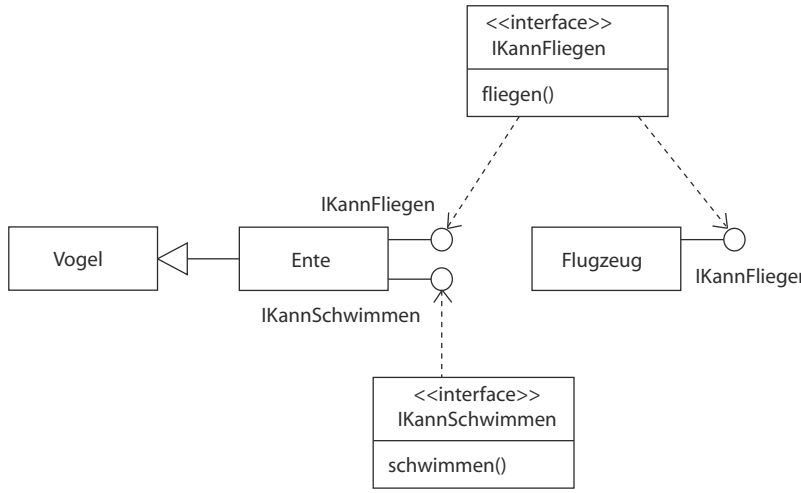


Abbildung 3.32
Beispiel Interface statt Vererbung

Um ein realistischeres Beispiel zu nennen: Das zuvor verwendete Codebeispiel in Listing 3.1 nutzt die Vererbung für ein polymorphes Verhalten zum Zeichnen beliebiger Formen. So weit, so gut, doch nun wollen Sie existierende Klassen verwenden, die Sterne und Spiralen zeichnen können. Dummerweise leiten sich diese bereits von einer anderen Oberklasse ab. Abhilfe schafft hier ein einheitliches Interface. Die existierenden Klassen für Sterne und Spiralen werden in eine umhüllende Klasse verpackt, welche das gemeinsame Interface implementiert und die bestehende, von der ursprünglichen Zeichnungsfunktion abweichende, Operation aufruft.

Die Trennung zwischen Schnittstelle und Implementierung ist Grundlage des *Dependency Injection*-Prinzips. Hier wird erst zur Laufzeit festgelegt, welche Komponente die Schnittstellen bedienen, gegen deren abstrakte Definition während der Entwicklungszeit programmiert wurde. Wir werden auf diesen Ansatz nochmals im Detail im Kapitel Pattern zurückkommen.

3.6 Anmerkungen

3.6.1 Textuelle Beschreibung

Diagramme sind eines, die sinnvolle Beschreibung das andere. Wie beim Kommentieren des Codes braucht es zur grafischen Darstellung textuelle Erläuterungen der Symbole. Andernfalls sind die Diagramme nichts weiter als Zeichnungen und meist nur für den Erstellter lesbar. Erst die Kombination von Text und grafischer Darstellung macht Modelle zu einem idealen Kommunikations- und Dokumentationsmittel. Der Fokus ist nicht auf die Suche nach der perfekten Abbildung unter Ausnutzung des gesamten Sprachumfangs von UML zu richten, sondern auf eine vollständige Beschreibung der für die jeweilige Betrachtung relevanten Elemente. Ihr Modell ist fertig, wenn jedes Element (Klasse, Verbindung, Interface, Operation, Attribut und

vieles mehr) entweder durch dessen Namen oder durch seine Beschreibung erklärt ist. Eine Zeichnung ohne Kommentar ist nutzlos, egal wie perfekt diese sein mag. Vermeiden Sie es jedoch, bei der Beschreibung Informationen zu wiederholen. Das Attribut `Vorname` der Klasse `Kunden` kommentieren zu wollen, macht wenig Sinn. Deshalb ist die richtige und sorgfältige Wahl der Bezeichnungen bereits die halbe Miete. Die Benennung sollte dabei nicht die Art und Weise der technischen Umsetzung wiedergeben, sondern deren Verantwortlichkeit hervorheben.

3.6.2 UML Sketch

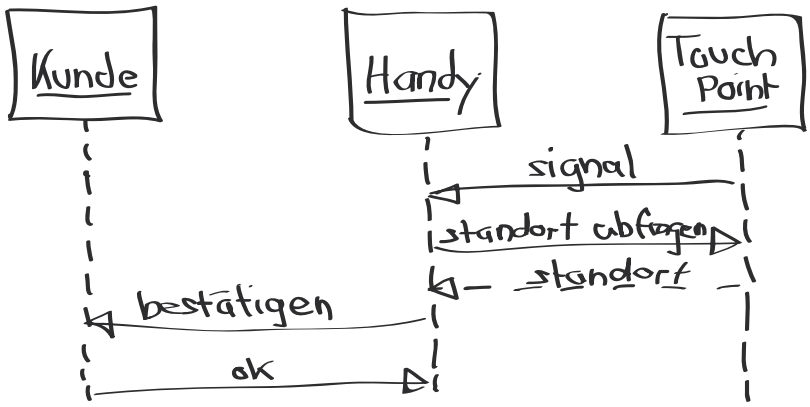


Abbildung 3.33
UML Sketch

UML ist eine Sprache. Eine Sprache, die dem gemeinsamen Verständnis dient und nicht zwingend eines Modellierungswerkzeugs bedarf. UML ist in erster Linie ein Kommunikationsmittel, dessen korrekte und vollständige Anwendung nicht an erster Stelle steht. Alle Methodiker werden mich für diese Aussage wohl mit Verachtung strafen. Dies soll kein Freibrief sein, die eigene UML-Notation zu erschaffen. Es ist eher ein Appell an den sinnvollen Einsatz und dabei den eigentlichen Zweck einer lauffähigen Software nicht aus den Augen zu verlieren. Oft scheitert der Einsatz von UML am Versuch, ein detailgetreues und vollständiges – nach Möglichkeit validiertes – Modell erstellen zu wollen. Innerhalb einer Arbeitsgruppe von Hand skizzierte Entwürfe dienen der Sache oft mehr als ein mit technischen Mitteln gefertigtes Modell, das nicht genutzt wird. Sind die Diagramme dann einmal im *Repository* erfasst, drucken Sie diese aus und kleben Sie damit Ihre Wand voll. Oft entstehen die besten Ideen, gemeinsam vor dieser Wand über die Konsequenzen der jüngsten Designentscheide zu diskutieren. Modelle sind praktische Hilfsmittel während der Entwurfsphase und keine einmal geschriebene, aber nie wieder gelesene Dokumentation. Dies funktioniert auch in einem formalen Prozess, wo die Erstellung schriftlicher Dokumente gesetzlich vorgeschrieben ist. Der beschränkte Platz einer Wand zwingt jeden zur Fokussierung auf die wesentlichen Dinge und die Modellierung der relevanten Aspekte. Die Präsentation des elektronisch erfassten Modells am Projektor ist nicht dasselbe.

3.7 Zusammenfassung

Eine Notation legt die Verwendung und Bedeutung von Symbolen fest, um Informationen schriftlich niederzuschreiben. Einer der umfangreichsten, heute als de facto geltender Standard für die Beschreibung von Softwaresystemen ist die Notation UML. Sie ist eine formale, durch ein Metamodell definierte Sprache, welche sich aus Dingen, Beziehungen und Diagrammen zusammensetzt. Jedes dieser Elemente hat eine bestimmte Bedeutung und ermöglicht es, ein System aus unterschiedlichen Perspektiven zu beschreiben. Diese Elemente lassen sich in die Gruppen Struktur, Verhalten und Interaktion aufteilen. Die Struktur schildert das System aus der statischen Perspektive und bedient sich der Klassen, Komponenten und Knoten. Das Verhalten beschreibt den dynamischen Aspekt eines einzelnen Dings der statischen Perspektive; es wirft einen Blick auf das Innenleben und die Funktionsweise. Die Interaktion zeigt das zeitliche Zusammenspiel und den Meldungsfluss der einzelnen Dinge wie Klassen und Komponenten auf, indem ein Szenario beschrieben wird.

UML legt keine Methode fest, wie eine Software zu modellieren ist und mit welchen Schritten eine Idee in ein fertiges Produkt transformiert wird. Sie definiert lediglich die Syntax und damit, wie diese Sprache einzusetzen ist. Es bedarf jedoch des gesunden Menschenverstands und einer Portion Pragmatismus, UML zweckmäßig einzusetzen. Die mit UML beschriebenen Modelle sind lediglich ein Hilfsmittel und Wissensspeicher, um einen für alle verständlichen Plan für den Bau einer Software zu entwerfen. UML bietet hierzu eine Vielzahl von Möglichkeiten, einen solchen Plan oder ein solches Modell zu visualisieren und mit standardisierten Symbolen zu erklären. UML ist für sich alleine für viele schwer verständlich, weshalb die Modelle entsprechend zu dokumentieren und zu ergänzen sind. In erster Linie zählt bei der Spezifikation die zu adressierende Leserschaft und weniger die formale Einhaltung einer bestimmten Notation.

3.8 Weiterführende Literatur

(Arlow & Neustadt, 2005) Das Buch von Arlow und Neustadt vereint eine ausführliche Diskussion über die Syntax von UML mit dem zweckmäßigen Einsatz in den entsprechenden Phasen und Disziplinen des iterativen Entwicklungsprozesses. Kein anderes Buch bietet eine solch vollständige und gut illustrierte Einführung in UML.

(Larman, 2005) Bereits in der dritten Auflage bietet das Buch eine umfassende Einführung in verschiedene Aspekte des Softwareentwurfs mit UML. Anhand der üblichen Phasen eines iterativen Prozesses in der Softwareentwicklung wird die Verwendung von UML für die jeweilige Aufgabe erklärt. Das Buch ist eine Fundgrube guter Ideen.

(Fowler & Scott, 2000) Auf wenigen Seiten vermag Martin Fowler auch in der dritten Ausgabe die essenziellen Punkte zu verdeutlichen und dabei eine

pragmatische Verwendung vorzustellen. Für alle, die dicke Bücher abschrecken, der richtige Einstieg in eine Vertiefung der hier dargestellten Konzepte.

(Oestereich, 2006) Im deutschsprachigen Raum seit vielen Jahren der Standard für eine Einführung in die Modellierungssprache UML. Dabei fokussiert das Buch auf die Anwendung von UML und weniger auf eine ausführliche Beschreibung der Sprache selbst.

(OMG, 2009) Sollte Ihnen die zuvor empfohlene Literatur nicht genügen oder es bleiben immer noch offene Fragen, so ruht die letzte Hoffnung auf der offiziellen Dokumentation des Standards UML auf mehr als 700 Seiten. Obwohl nicht gerade ein Lesevergnügen, bietet sie Antworten auf viele Detailpunkte.