

Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2016 Beuth Hochschule für Technik Berlin

UML - Unified Modeling Language



Überblick und Lernziele



Lernziele

Ziel dieser Lerneinheit ist es, sie mit der UML vertraut zu machen. Dazu werden - neben der Einführung - statische Strukturdiagramme und dynamische Diagramme (Verhaltensdiagramme) vorgestellt.

Sie sollen den geeigneten Einsatz von UML beurteilen können und UML praktisch an einem eigenen Projekt anwenden. Dabei sollen die verschiedenen Elemente von UML genutzt werden sowie die Vor- und Nachteile dieser Darstellungsform erkannt und beschrieben werden. Die kritische Nutzung dieser Industriesprache sollte auf jeden Fall beherrscht werden.

Als Ergebnis der praktischen Übung sind Sie in der Lage beurteilen zu können, welche UML Diagramme sie in welcher Reihenfolge anwenden, um ihr Modellierungsziel zu erreichen.



Gliederung der Lerneinheit

- Einführung
- Hintergrund und Historie der UML
- Vorstellung statischer Strukturdiagramme
- Vorstellung dynamischer Diagramme
- Wissensüberprüfung und Übungen



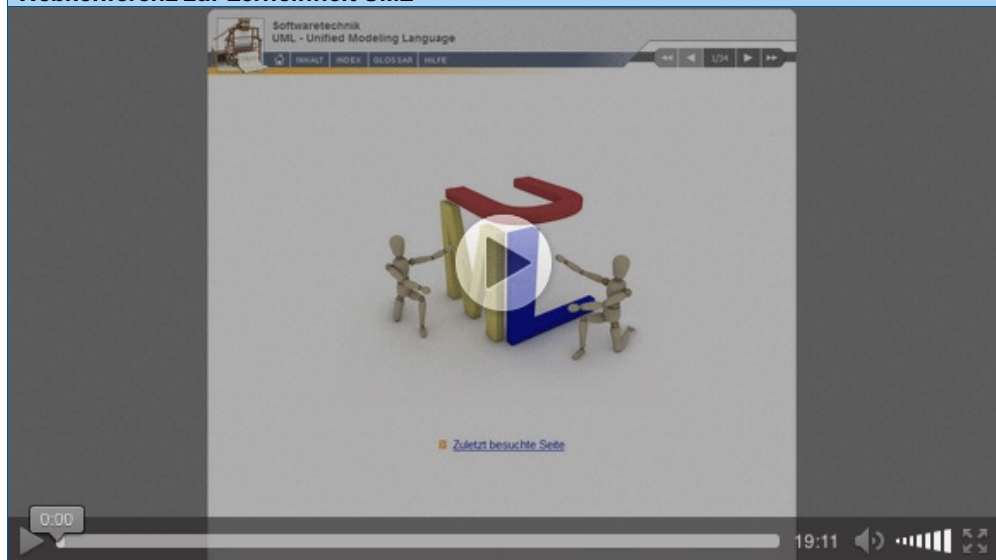
Zeitbedarf und Umfang

Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 120 Minuten und für die Bearbeitung der Übungen ca. 170 Minuten.



Film

Webkonferenz zur Lerneinheit UML



© Beuth Hochschule Berlin - Dauer: 19:11 Min. - Streaming Media 25.3 MB


Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

1 Unified Modeling Language



Definition

Die **Unified Modeling Language** ist eine Menge von Notationselementen, mit denen Modelle für Softwaresysteme entwickelt werden können. Dies betrifft die Analyse, das Design und ganz allgemein die Darstellung und Dokumentation der Softwareelemente oder des Softwareverhaltens.

Die UML wurde von der  **OMG** entwickelt und ist in über hundert Seiten spezifiziert. Sie hat sich als wichtigste Notation / Sprache zur Spezifikation von Softwaresystemen durchgesetzt. Eine der wichtigsten Aufgaben ist es, den Entwurf und die damit verbundenen Architekturen für andere Projektbeteiligte oder außenstehende Personen transparent zu machen. Wichtig an UML ist, dass es nicht absolut dogmatisch angewendet werden muss.

Ziel von UML-Diagrammen oder Beschreibungen ist im Wesentlichen, dass andere Personen die Aussage verstehen. Es gibt also kein Gremium, welches über richtiges und falsches UML entscheidet. UML selbst kann an vielen Stellen durchaus interpretiert werden.

Es ist sicher besser Energie in **gutes Design** zu stecken, als in **100%korrektes UML**. Die Verwendung von Darstellungen und Methoden, die nicht UML sind, sind dafür wichtig. Beispielsweise werden häufig Entscheidungstabellen für Sachverhalte benötigt. Diese sind nicht Teil der UML, aber äußerst hilfreich.

Experimentieren Sie mit UML genauso wie mit Non-UML und sprechen Sie immer wieder mit Leuten, die schon Erfahrung auf dem Gebiet gemacht haben. Viele Architekten sind beispielsweise der Meinung, dass eine gute Designsession am Whiteboard ohne striktes UML mehr bringt, als viele UML-Diagramme auf Papier.

Randbemerkung

Die **OMG** verwaltet noch weitere interessante Standards wie CWM, MOF und das für UML wichtige Austauschformat XML.

1.1 Literatur



Hinweis

Beim Design, bei der Arbeit mit UML-Werkzeugen und in der Projektarbeit ist es ungemein wichtig gute Literatur zu verwenden, um die Feinheiten und die Bedeutung der Notationselemente richtig nachschlagen zu können. Die hier angegebenen Bücher beinhalten auch viele Beispiele.

Drei Werke sollen hier besonders erwähnt werden, die auch bei der Erstellung des Studienmoduls hilfreich waren:

- **MARTIN FOWLER:** UML Distilled, Pearson Publishers
- **JECKLE, RUPP, HAHN, ZENGLER, QUEINS:** UML 2 glasklar, Hanser
- **BERND OESTERREICH:** Analyse und Design mit UML 2.1, Oldenbourg

1.2 Werkzeuge

Damit Sie auch gleich alles praktisch ausprobieren können, benötigen Sie ein UML-Werkzeug.

Unter der Adresse <http://www.jeckle.de/umltools.htm> sind weit über hundert UML-Werkzeuge zu finden. Auch die in Deutschland bekannte Firma OOSE hat eine gute [Werkzeugseite](#). Es wird dabei einige Zeit brauchen, bis Sie „Ihr“ Werkzeug gefunden haben. Planen Sie dafür Zeit ein! Oftmals verklemmen sich irgendwelche Einstellungen, und die Werkzeuge zeigen nicht das was Sie wünschen oder generieren sogar fehlerhaften Programmcode.



Hinweis

An dieser Stelle können lediglich ein paar Tipps gegeben werden, welche Tools sich hier und da schon einmal bewährt haben:

- **Together** www.borland.com - Als mächtiges und flexibles Werkzeug bekannt
- **Omondo** www.ejb3.org - Sehr schön in Eclipse integriert
- **Rational Rose** www.ibm.com - Nicht kostenfrei, mächtig, in die rational Toolsuite integriert, hat den Ruf etwas klobig zu sein
- **Magic Draw** www.nomagic.com - Kostenpflichtig
- **Argo UML** www.argouml.tigris.org
- **BlueJ** www.bluej.org - Ist als Lehrwerkzeug hervorgegangen, ist mit dem Development eng verzahnt und kann nur Klassendiagramme
- **Fujaba** www.fujaba.de/ - Ein bekanntes freies Werkzeug der Uni-Paderborn
- **Astah** www.astah.net/editions/professional - Kommt aus Japan
- **ObjectlF** www.microtool.de von Microtool aus Berlin!
- **Innovator** www.mid.de verschiedene Ausprägungen für diverse Sichtweisen (Db, Architekt, Entwickler etc.)
- **Enterprise Architect** www.sparxsystems.com Sehr mächtiges und vergleichsweise preisgünstiges Modellierungstool für UML und andere Sprachen.

Weitere Empfehlungen

www.umlet.com

www.polarsys.org

1.3 Diagrammübersicht

Strukturdiagramme und Architektordiagramme werden auch als statische Diagramme bezeichnet. Verhaltensdiagramme und Interaktionsdiagramme werden oft auch als dynamische Diagramme bezeichnet.

Strukturdiagramme

Paketdiagramm (Package)	Zeigt die konkrete Struktur der Software in Namespaces oder Paketen mit deren Abhängigkeiten.
Klassendiagramm (Class)	Stellt die Klassen der jeweiligen Programmiersprache mit deren Beziehungen dar.
Objektdiagramm (Object)	Zeigt Objekte - also in welchem Zustand die Klassen beispielsweise vorkommen können.

Tab.: Strukturdiagramme

Architektordiagramme

Einsatz- und Verteilungsdiagramm (Deployment - Diagram)	Zeigt, wo die (meist Hardware-) Systeme und ihre Komponenten in ihren Teilen auf Rechnerknoten installiert sind.
Komponentendiagramm (Component - Diagram)	Zeigt die Komponentenstruktur mit den Beziehungen untereinander.
Kompositionsstrukturdiagramm (Composit-Structure - Diagram)	Zeigt die Komponenten mit ihren ganz konkreten ex- und importierten Schnittstellen.
Subsystemdiagramm	Architekturzusammenhänge von Komponenten.

Tab.: Architektordiagramme

Verhaltensdiagramme

Anwendungsfalldiagramm (Use-Case-Diagram)	Stellt die Akteure mit ihren Anwendungsfällen (quasi Wünschen) dar.
Aktivitätsdiagramm (Activity- Diagram)	Zeigt, wie der Ablauf des Programmes ist, indem Aktionen, Übergänge (Transitionen) und Verzweigungen verwendet werden.
Zustandsdiagramm (State- Diagram)	Hier werden Objektzustände dargestellt. Zwischen den Zuständen gibt es Übergänge. Ein Automat kann die Reihenfolge der Übergänge noch genauer spezifizieren.

Tab.: Verhaltensdiagramme

Interaktionsdiagramme

Sequenzdiagramm (Sequence - Diagram)	Stellt dar, wie Klassen oder Komponenten mittels Nachrichten interagieren / kommunizieren.
Interaktionsübersicht (Interaction Overview)	Eine Kombination aus Aktivitäts- und Sequenzdiagramm.
Kommunikationsdiagramm (Communication)	Ordnet die ausgetauschten Nachrichten graphisch an, um sie geeignet darzustellen.
Zeitdiagramm (Timing- Diagram)	Beschreibt den zeitlichen Ablauf der Objektzustände (oft in der Telekommunikation und Steuerungs- / Regelungstechnik verwendet).

Tab.: Interaktionsdiagramme

2 Historie

Die Beschäftigung mit der Thematik der objektorientierten Analyse / Design kam erst in den 90er Jahren in Schwung. Autoren, die sich hier früh einen Namen gemacht haben waren **ADELE GOLDBERG**, **GRADY BOOCH**, **PETER COAD**, **EDWARD YOURDON**, **JAMES RUMBAUGH**, **BERTRAND MEYER**, und **IVAR JACOBSEN**.

Schon früh setzten sich die Methoden von **JAMES RUMBAUGH** und **GRADY BOOCH** durch. Beide konkurrierten jedoch. Rumbaugh's Methode schien besser für die Analyse zu sein und Booch's Methode war wohl beim Design vorteilhafter. **JAMES RUMBAUGH** wurde zudem 1994 von der Rational Software Cooperation eingestellt, und war dort auch für die Weiterentwicklung der IBM Toolsuite verantwortlich.

Mit den Kräften von **IVAR JACOBSON** vereint, versuchten **RUMBAUGH** und **BOOCH** dann ihre Differenzen zu überbrücken und eine einheitliche Sprache zu erschaffen. Dies natürlich auch etwas auf Druck der Industrie hin. Denn schließlich ließ sich mit entsprechenden Werkzeugen und Vorgehensmodellen auf Basis von UML durchaus Geld verdienen.

IVAR JACOBSENS Firma Objectory wurde dann auch von Rational gekauft und aus der Vereinheitlichung der Methoden entstand die Unified Method (UM).

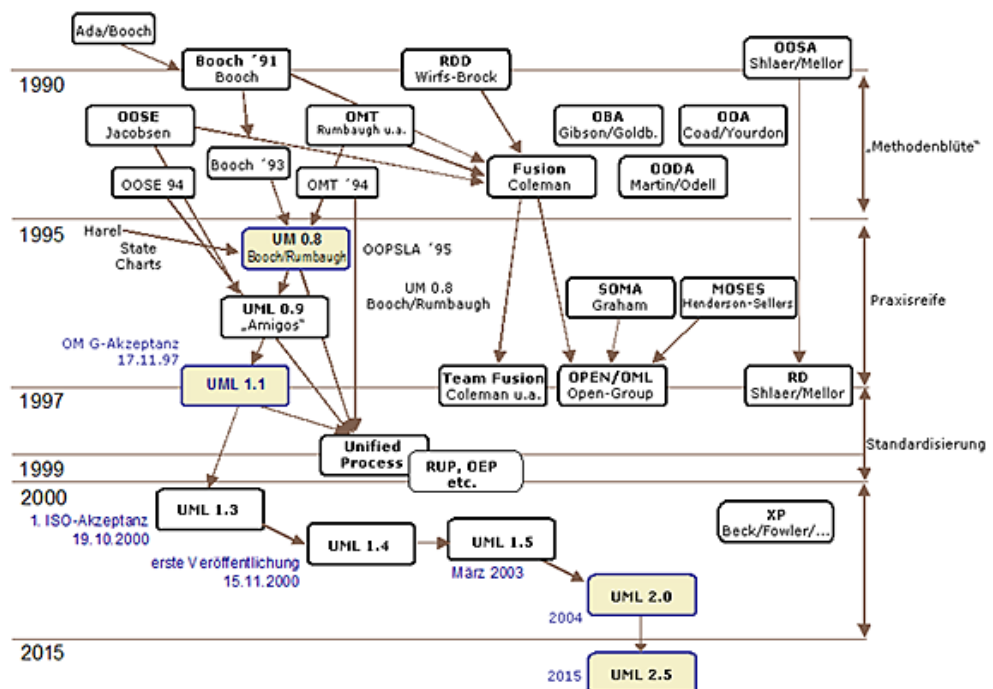



Abb.: Entwicklung der UML

BOOCH, **RUMBAUGH** und **JACOBSENS** wurden fortan die drei Amigos genannt. Aus der Zeit ihrer früher heftigen Kämpfe um Methoden, des Diskussionsprozess und der schwierigen Verhandlungen ist ein selbstkritischer Ausspruch von **JACOBSENS** bekannt, der immer wieder gerne zitiert wird:

*„What's the difference between a terrorist and a methodologist?
You can negotiate with a terrorist!“*

Der Druck wurde jedoch größer, eine nicht proprietäre Sprache auf Basis der UM zu entwickeln: Die Unified Modeling Language (UML). Ein internationales Konsortium entwickelte 1996 die UML und übergab sie dann an die **OMG**. So entstand die UML 1.1 im Jahre 1997. Es entwickelte sich dann in Schritten zur Version 2.0, die bis 2006 die offiziell letzte downloadbare Version war. Momentan ist die Version 2.5 vom Juni 2015 im Downloadbereich der **OMG** verfügbar.

2.1 UML Struktur

Die aktuellste UML-Spezifikation kann von der Seite der  **OMG** heruntergeladen werden - auch ältere Versionen sind dort verfügbar. Es lohnt sich, einmal kurz etwas auf den UML Webseiten herumzustöbern. Die Spezifikation gliedert sich in die folgenden Teile:



1. **UML 2.0 Infrastructure Specification:**

Hier werden Basiselemente der UML vorgestellt, die Voraussetzung für die nachfolgenden Spezifikationen sind. Elemente wie Klasse, Assoziation oder Multiplizität werden vorgestellt.

2. **UML 2.0 Superstructure Specification:**

Hier werden die eigentlichen bekannten Elemente der UML definiert und mit Anwendungsfällen und tieferliegenden Konzepten vorgestellt.

3. **UML 2.0 Object Constraint Language:**

Das letzte Dokument spezifiziert Randbedingungen für einige Diagrammart. Sie ist z. B. für Klassendiagramme ( Kap. 4.4) oder Sequenzdiagramme ( Kap. 5.4) wichtig. Dort können mit Invarianten, Pre-/Post-Conditions oder Guards Einschränkungen angegeben werden. So ist es beispielsweise sehr hilfreich, für Methoden Vor- und Nachbedingungen anzugeben. Oder für Felder gleich Randbedingungen zu notieren (z. B. darf nie < 0 sein). Diese Randbedingungen sind dann z. B. auch für die maschinelle Weiterverarbeitung der UML wichtig.

4. **UML 2.0 Diagram Interchange:**

Ein vierter Teil spezifiziert das Layout der Diagramme. Mit dieser Spezifikation soll mit dem werkzeugübergreifenden Austausch von Diagrammen nun auch das Layout nicht mehr verloren gehen, da jedes Diagramm mittels eines Graphen (Knoten, Kanten, Blätter) beschrieben wird.

2.2 Wie nutzt man UML

UML mag eine schöne Notation sein, aber einigen dürfte noch nicht so ersichtlich sein, welche Einsatzgebiete es überhaupt gibt.

Kommunizieren und Abstrahieren

Der eigentliche Sinn der UML liegt darin, **Ideen zu kommunizieren!**

Die Frage ist immer, wie kann man am besten und schnellsten das Design eines Systems herüberbringen und es dokumentieren. Es ist klar dass UML vielleicht nicht immer das beste Mittel sein muss und auch nicht vollständig sein kann.

Daher gibt es auch viele Kritiker die sagen, dass UML zu groß und zu komplex sei.

UML versucht es allen Personen recht zu machen. Trotzdem hat sich UML gegenüber all seinen vorangegangenen Notationen etabliert. Es ist die Sprache, die sich in der Industrie durchgesetzt hat und welche Sie auf jeden Fall beherrschen, dennoch kritisch betrachten sollten.

Ein weiterer wichtiger Nutzen der UML liegt darin, **Details zu verbergen** und später dann diese mit oder ohne UML verfeinern zu können. In vielen Fällen müssen Sie sich dem System Top-Down nähern und aus der Vision / Analyse konkrete Teile oder Komponenten spezifizieren.

Wie gelingt ihnen dies? Hier können UML-Diagramme helfen, die grobe Struktur / Anforderungen zu definieren und diese dann in feineren Diagrammen festzuschreiben. Oftmals müssen Sie aber auch einfach mit anderen Designern, Analysten, Stakeholdern, Projektleitern oder Auftraggebern sprechen, die manchmal von IT keine Ahnung haben. Wie stellen Sie denen das System dar? Auch wenn viele der Personengruppen keine umfassenden IT-Kenntnisse haben, so haben die meisten dennoch einen Einblick in UML und sind mit den Bedeutungen einiger Diagramme vertraut. Die Bedeutung eines Use-Case- oder Deploymentdiagrammes werden Sie bald gut genug erklären können und somit die Grundstruktur ihres Systems vermitteln können.

• Reverse-Engineering

Eine - bisher wenig dokumentierte - Anwendungsmöglichkeit der UML ist das Reverse-Engineering. Hierbei haben Sie vielleicht schon ein System, das weiterentwickelt werden soll und Sie möchten sich einen Überblick über das System verschaffen. Gute Werkzeuge können den bestehenden Code analysieren und daraus verschiedene UML-Diagramme generieren. Es kann sehr hilfreich sein, sich aus einem Berg von Code erst einmal einige Diagramme (z. B. Klassen-, Package- oder Komponentendiagramme) zeigen zu lassen und so das Programm erst einmal „meta“ zu verstehen.

Reverse Engineering ist ein nicht zu unterschätzender Ansatz in der Industrie, wenn es darum geht Wettbewerbsvorteile zu erreichen. Ein bekanntes Beispiel für Reverse Engineering ist die Tabellenkalkulation Lotus 1-2-3, das Programm wurde von Microsoft „zerlegt“, „verstanden“ und als MS Excel neu gebaut. Auch das ist Reverse Engineering.

• Sketch- / Blueprint-Mode

Neben der schriftlichen Dokumentation einer Architektur von Architekten wird UML oft mehr informell in Teamdiskussionen quasi als Sketch verwendet. Im Sketch Mode wird informelles UML verwendet und die dargestellten UML Diagramme zeigen meist nur die interessanten Ausschnitte des Systems, das gerade besprochen werden soll.
(sketch (engl.) = „Skizze“)

Im erstgenannten Blueprint-Mode ist meistens der Architekt des Systems damit beschäftigt, das Design der Software mehr oder weniger vollständig zu hinterlegen (blueprint = Blaupause; ein Begriff aus der Architektur).

Sketch- und Blueprint-Mode sind nicht synonym zu verwenden. Sketch ist eine kleine Diskussion über Ausschnitte. Blueprint ist eine größere Architekturvorgabe.

• Programmiersprache / MDA

Ein wichtiges Element der UML ist, dass es selbst von Maschinen weiterverarbeitet werden kann. Die UML und die dazugehörigen Sprachen wie XML (XML Metadata Interchange) werden von Frameworks gelesen, die dann auch Code generieren können.

Das Entwickeln von Software ist relativ teuer, gerade europäische IT-Firmen müssen sehen, wie Sie konkurrenzfähig und schnell, qualitativ (!) hochwertige Software erstellen können.

Dies kann sicher teilweise dadurch geschehen, dass Code automatisiert generiert wird. Wenn UML Diagramme die Komponenten, Pakete, Klassen und auch noch das Verhalten von Software beschreiben, wieso soll man dann nicht möglichst viel Code generieren lassen? In diesem Fall wäre dann die UML selbst so etwas wie eine höherwertige Meta-Programmiersprache.

Dieses Vorgehen führt uns zum Forschungsgebiet der **MDA** (Model Driven Architecture), wofür es bereits erstaunlich viele leistungsfähige Frameworks gibt. Leider sind viele dieser Toolkits in der Industrie versteckt. Es gibt aber auch einige gute freie Toolkits z. B. unter www.AndroMDA.org.

MDA ist Bestandteil des Studienmoduls „Modellbasierte Softwarekonstruktion“ und wird dort intensiver behandelt.

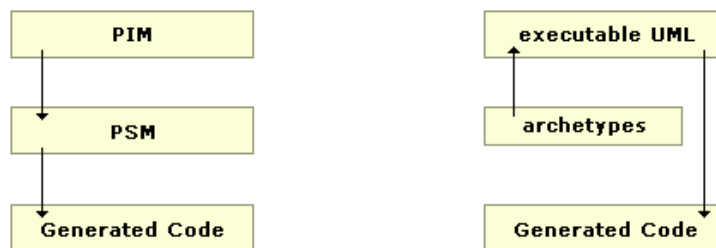


Abb.: Model Driven Architecture (MDA)

Der wichtigste Zweig ist links dargestellt. Sie modellieren in UML. Daraus können Sie gegebenenfalls ein Modell erstellen - das **Platform Independent Model (PIM)** - welches konkreter ist, aber noch unabhängig von der darunterliegenden Technologie.

Im darauf folgenden Schritt würden Sie (alles mit MDA Tools) ein **PSM (Platform Specific Model)** generieren, welches konkreten und lauffähigen Code für eine Plattform enthält. Z. B. Code für eine .NET oder J2EE Plattform. Das bedeutet es gibt tatsächlich Frameworks, die ihnen lauffähigen Code generieren können. Manchmal sogar lauffähige Anwendungen, die viel mit Templates oder vorgefertigten Bausteinen arbeiten.

Ein weiterer Ansatz ist **executable UML**. Hier wird der Zwischenschritt - des zu erstellenden PSM - übersprungen und der Code gleich anschließend generiert. Archetypen helfen hier zu beschreiben, wie der Kodierungsschritt von UML zum Code geschehen soll.

2.3 Designablauf

Bei der Auseinandersetzung mit Design besteht die anfängliche Schwierigkeit oftmals darin, zu wissen wie das konkrete Vorgehen aussieht. Dabei ist vorrangig zu klären, welches Diagramm als erstes gewählt werden soll und warum.

*Welche **Reihenfolge** ist am besten?*

Natürlich ist diese Frage schwer zu beantworten und hängt zu 100% vom zu designenden System ab. Dennoch soll hier mal ein Vorschlag gewagt werden, der auf Projekterfahrung basiert und den Abstraktionsgrad der Diagramme mit berücksichtigt (hier einmal mit englischem Namen und „D“ für Diagramm):

1. Use Case D

Die Anwendungsfälle sind sicher mit am abstraktesten. Was der Anwender vom System will sollte in der Regel zuerst bekannt sein und gehört ja schon eher in die Analyse als ins Design.

2. Deployment D

Da hier gezeigt wird, wo welche grobe Komponente läuft ist dies auch sehr abstrakt. Hier geht es quasi um physische Locations bei großen Projekten. Also Server X hier, viele Clients da, ein Abrechnungsserver dort, etc. Das ist abstrakt und gehört in der Regel weit nach vorne.

3. Paket D

Die Pakete beinhalten meistens Namespaces für die Sammlung von Komponenten und Klassen. In der Regel wird dies auch früh im Projekt definiert, damit Developer schon mal eine Spielwiese haben um Dinge zu testen, bevor es richtig losgeht. Hier werden auch indirekt Zuständigkeiten definiert.

4. Component D

In Paketen sind meistens Komponenten zu finden, welche größtenteils aus einer Sammlung von Klassen bestehen. Daher müssen Komponenten-Diagramme vor Klassen-Diagrammen angelegt werden.

5. Class D

Aus diesem Grunde sollten Klassen-Diagramme nach Packages und Komponenten-Diagrammen erstellt werden.

6. [Composite-Structure D, Subsystem D, Object D]

Die hier genannten Diagramme werden gar nicht so häufig verwendet und oft in der gesamten Design Phase an beliebiger Stelle mit eingestreut.

7. Activity D

Die Aktivitäten einer Anwendung werden in der Regel viel früher in dieser Reihenfolge spezifiziert, damit die Ablaufstruktur der Anwendung klar wird. Dies kann oft die Voraussetzung dafür sein, damit dem Designer die Komponenten klar werden. Manchmal sogar noch vor dem Deployment Diagramm aber nach den Use-Cases. Der Übersichtlichkeit halber ist es hier nach den statischen Diagrammen bei den dynamischen Diagrammen angesiedelt.

8. Sequence D

Sie zeigen das Verhalten von Komponenten bzw. Klassen. In der Regel müssen also erst Komponenten oder Klassen vorhanden sein, damit das Sequenzdiagramm modelliert werden kann.

9. State D

In welchem Zustand sich die Anwendung, die Komponenten oder Klassen befinden kann ebenfalls jederzeit in der Designphase modelliert werden. In der Regel nachdem die Use-Cases, Deployment und Activity-Diagramme erstellt wurden.

10. [Interaction-Overview D, Timing D]

Diese nicht so häufig verwendeten Diagramme können auch bei Bedarf überall eingesetzt werden.

Ideal wäre es also jetzt, wenn sie sich ein eigenes motivierendes (privates) Projekt nehmen und dieses modellieren und über einen längeren Zeitraum begleiten. Vielleicht helfen dabei die obenstehende Orientierung im Designablauf und die in Kapitel 4 und 5 folgenden Erläuterungen der konkreten Diagramme.

3 UML Bewertung

Die Bedeutung von UML ist in der Realität noch viel umstrittener, als es den Anschein hat.

Im akademischen Bereich wird UML an vielen Stellen gelehrt, da es noch viele Befürworter auf Seiten der Hochschule gibt. Nicht selten wird dies dann auch in der Industrie unkritisch übernommen.

Dennoch gibt es Unternehmen und Bewegungen die die Nützlichkeit von UML viel geringer bewerten. Gerade auch im agilen Umfeld wird deshalb oft auf andere Techniken gesetzt. Dies ist bereits aus dem mittlerweile recht „alten“ XP Modell zu sehen, welches auch keine umfangreichen UML Artefakte vorsieht. **In einigen bedeutenden Unternehmen wie bspw. Google ist UML sogar verboten!**

Das soll nicht heißen, dass UML vollständig untersagt ist. Wenn zwei Entwickler sich unbedingt mit UML „unterhalten“ möchten, so ist dies kein Problem. Jedoch wird die Kraft eines Prototypen oder eine prototypische Architektur in der Unternehmenskultur als viel effizienter angesehen als ein UML-Papier.

Entscheidend für jeden Softwaretechniker ist daher, das Spektrum der Werkzeuge oder Kommunikationsmittel zu kennen und selbst beurteilen zu können, inwieweit UML ein geeignetes Mittel ist oder ob andere Verfahren besser geeignet sind.

4 Statische Strukturdiagramme

In diesem Teil werden die statischen Strukturdiagramme vorgestellt:

- Verteilungsdiagramm: Wie sind die Softwarekomponenten physisch verteilt?
- Komponentendiagramm: Aus welchen Einheiten / Komponenten besteht das Softwaresystem?
- Paketdiagramm: Welche konkreten Pakete / Namensräume werden genutzt, um die Komponenten zu hierarchisieren?
- Klassendiagramm: Welche Klassen gibt es (in einem Namensraum)?
- Kompositionsstrukturdiagramm: Wie hängen Komponenten genauer zusammen?
- Objektdiagramm: Welche Objekte gibt es zu einem bestimmten Zeitpunkt?

4.1 Verteilungsdiagramm

Das Verteilungsdiagramm (engl. Deployment Diagram) wird relativ früh spezifiziert. Relativ bald nach den Use-Cases. In diesem Diagramm geht es darum, die physische Struktur der gesamten Anwendung zu zeigen. Also welche Komponenten an welchem Ort auf welchen physischen Geräten laufen.

Dies ist für Studenten oft schwer nachzuvollziehen, da man in der Regel nur kleine Projekte schreibt, die nur auf einem Rechner laufen.

Beispielsweise gibt es im Projekt TollCollect verschiedene Rechenzentren, Satelliten, Geräte auf Autobahnen und Geräte in den Raststätten. Es läuft also ganz unterschiedliche Software auf sehr verschiedenen Rechnern an vielen Stellen.

Das Ziel des Verteilungsdiagrammes ist es, die physische Verteilung schlüssig darzustellen. Dazu ein Beispiel:



Beispiel

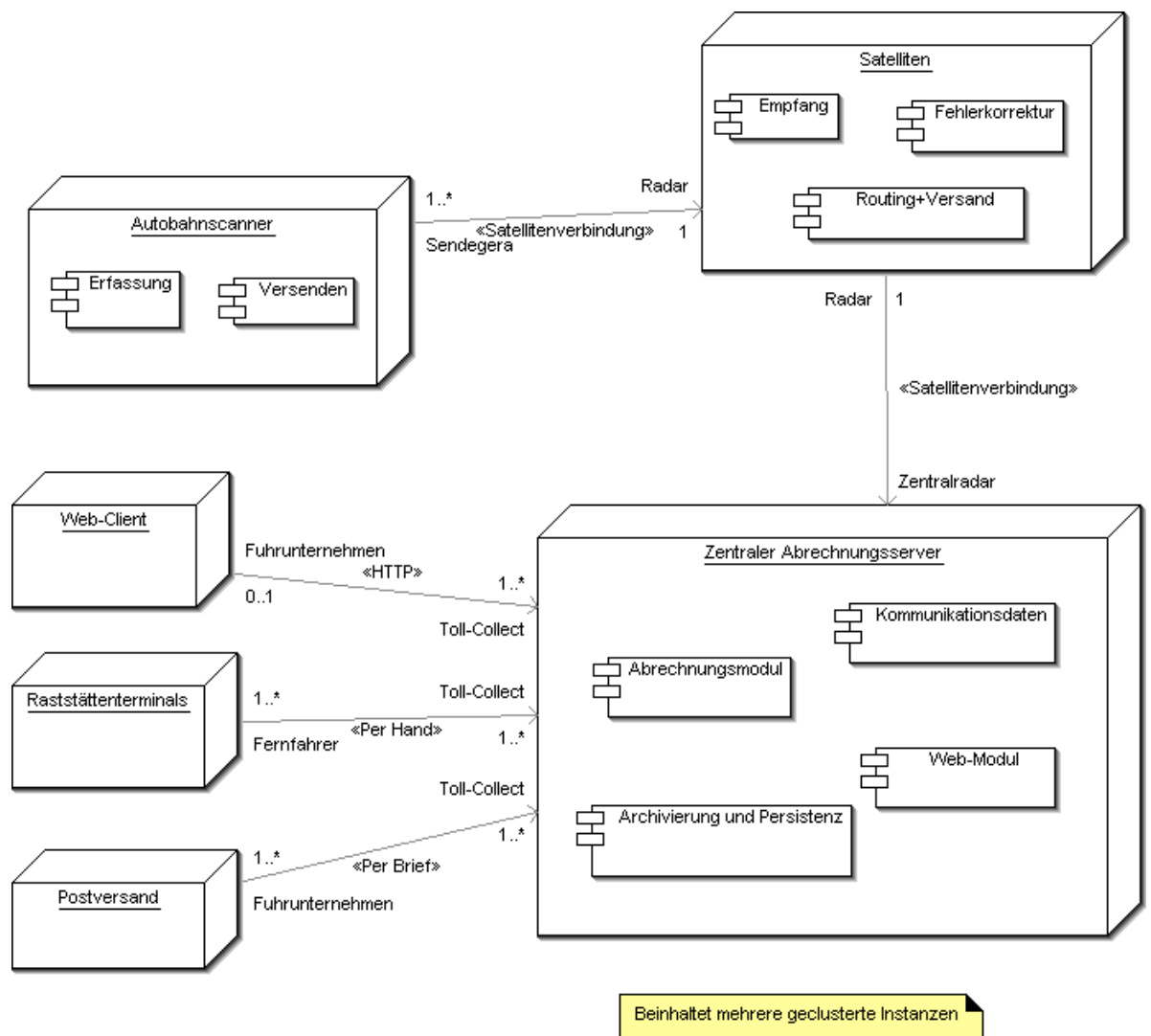


Abb.: Verteilungsdiagramm des Systems Toll-Collect

Wir sehen hier ein Beispiel für das Maut System Toll-Collect. LKWs werden an den Autobahnen erfasst. Diese Daten werden dann per Satellit an die Zentrale gesendet. Dort zeigen die Komponenten im Zentralsystem, welche Aufgaben die Komponenten wahrnehmen. Für die Bezahlung beispielsweise können die Anwender entweder an die Raststätten gehen oder per Web oder per Post bezahlen. Dieses Beispiel soll Ihnen helfen, die physische Verteilung von solch großen Systemen nachvollziehen zu können.

Notation

Die einzelnen physischen Locations / Knoten (ggf. Server) werden durch Kästen / Quader dargestellt. Die Kästen werden durch einfache Linien miteinander verbunden und stellen die Kommunikation untereinander dar. In den Kästen können Komponenten oder Artefakte dargestellt werden. Artefakte können beispielsweise Dateien unterschiedlicher Formate wie, **jar**, **dll**, **war** sein.

Die Verbindungen der Knoten beschreiben die Art der Kommunikationsverbindung. Weiterhin können Multiplizitäten angegeben werden. Beispielsweise kann man angeben, das sich bis zu 1000 Clients gleichzeitig mit dem Server verbinden dürfen.




Hinweis


1. Zur Übung empfiehlt es sich, einfach mal ein größeres System zu modellieren, als das, was man später dann bauen oder feiner spezifizieren möchte. Dadurch erreicht man ein umfangreicheres Verteilungsdiagramm. Oder planen Sie Ihre Anwendung gleich als komplexeres verteiltes System.
2. Stellen Sie sich vor, der Systemadministrator kommt zu Ihnen - er weiß nur, es kommt Arbeit auf ihn zu - und möchte wissen, welche Hardware er bereitstellen muss. Dabei hilft ihm das Verteilungsdiagramm. Mit den zusätzlichen Komponenten darin hat er auch die Möglichkeit zu verstehen, was das System grob tun soll und kann auch gleich die Kosten abschätzen.

4.2 Komponentendiagramm

Das Komponentendiagramm (engl. Component Diagram) beschreibt eine bestimmte Menge von Komponenten. Das Komponentendiagramm ist die Antwort auf die Frage, wie das konkrete System strukturiert ist.

In der Regel sind es mehrere Klassen, es kann aber auch nur eine Klasse sein, die aber keinesfalls Details zeigt. Schon im  Deployment-Diagramm war eine Menge von Komponenten zu sehen, die einzeln oder zusammen eine oder mehrere bestimmte Aufgaben erfüllen.

Wichtig dabei ist, dass man Abhängigkeiten der Komponenten spezifizieren kann und auch sollte. Komponenten bieten Schnittstellen, die von anderen Komponenten genutzt werden können.

BERND ÖSTERREICH schreibt dazu: „Eine Komponente ist ähnlich wie eine Klasse instanzierbar und kapselt komplexes Verhalten. Mit ihr werden Einheiten gebildet, die eine hohe fachliche Kohärenz haben. Im Gegensatz zu einer Klasse wird bei einer Komponente zusätzlich auch die prinzipielle Austauschbarkeit (Substituierbarkeit) angestrebt. Eine technische Plattform für Komponenten ist beispielsweise Enterprise Java Beans (EJB).“  [Oes98]

Notation

Die Komponente ist ein Rechteck mit dem typischen „Lego-Symbol“ in der rechten oberen Ecke. Es enthält das Schlüsselwort `<<component>>`, das aber oft der Übersichtlichkeit halber nicht mit angezeigt wird.

Es ist erlaubt weitere Elemente - wie Objekte oder Komponenten - in das Objekt einzufügen, was aber leicht zu sehr komplexen Diagrammen führen kann (**packable Element**).

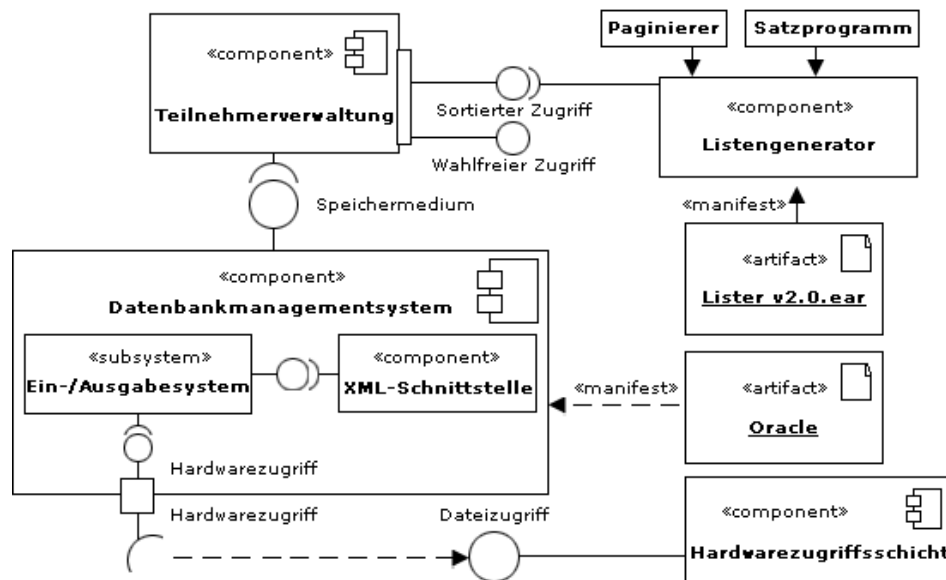


Abb.: Beispiel
Komponentendiagramm

Was zeigt dieses Diagramm?

- Einige Komponenten (mit dem Lego-Symbol)
- Rechts einige **Artefakte**. Dies kann Papier oder Software sein.
- Interessant sind dann noch die **Kugeln**, in die der Halbkreis greift. Die Kugel ist die bereitgestellte Schnittstelle. Der Halbkreis die benötigte Schnittstelle.
- Mit kleinen Quadraten gibt es noch **Ports** die einen Zugriff auf die Komponente realisieren, der nicht quasi auf Interfaces basiert.

In Diagrammen geistern aus verschiedenen UML-Variationen noch die Schlüsselwörter `<<realize>>` oder `<<reside>>` (von welcher Klasse wird diese Komponente realisiert) oder `<<implements>>` herum. Letzteres beschreibt das konkrete Interface, das diese Komponente auch implementiert.



Hintergrundwissen

Wieder kann man sich vorstellen, dass dieses Diagramm nicht nur zur internen Kommunikation dient, sondern auch dem Dialog mit Entwicklern und Auftraggebern.

Ein paar Beispiele:

- Der Auftraggeber will wissen, wie die neuen Komponenten mit den Alten zusammenspielen.
- Der Auftraggeber möchte wissen und dann selbst entscheiden, welche Komponenten (oder Module) er wann kaufen kann und wie diese dann zusammenpassen.
- Der Junior Entwickler möchte von Ihnen wissen, in welche Komponenten er das z. B. Volltextindizierungssystem zerlegen soll. Er fragt, wie das geht, wie man das baut. Sie zeichnen ein Diagramm mit Komponenten, wie beispielsweise Parser, Konfiguration der Search-Engine, Konfiguration der externen Module, Aktivierung der Engine, Handling und Aufbereitung der Ergebnisse, .etc. Ergebnis: Der Junior Entwickler versteht und hat eine Vision der Struktur.

Oftmals spiegeln sich hier Designpatterns wie **MVC** wieder.



Wichtig

Wichtig ist in diesem Zusammenhang, dass es in UML auch noch den Stereotyp **Subsystem** gibt, der häufig in Komponentendiagrammen angewendet wird. Es wird hier quasi nur ein Container für andere Komponenten aufgebaut - dargestellt als einfacher Kasten mit dem Stereotyp **Subsystem**.

4.3 Paketdiagramm

Das Paketdiagramm strukturiert den Namensraum von Paketen, Komponenten oder Klassen.

Es korrespondiert üblicherweise mit den Packages in Java oder mit den Namespaces in C#. Beispielsweise `de.vfh.portal.datenbank` in der die Klasse `Zugriffsschicht` enthalten sein kann. Wichtig dabei ist eine hierarchische Struktur, welche die eindeutige Identifizierung des Modellelementes ermöglicht.

Paketdiagramm

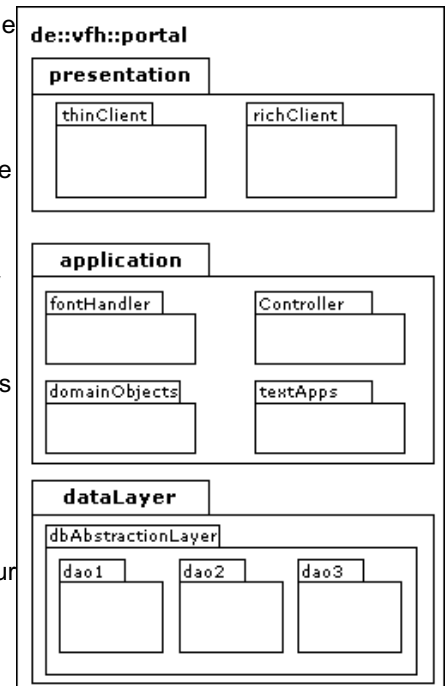
(Siehe Anhang) Im Beispiel rechts erkennt man, dass alle gezeichneten Pakete sich unter dem Namespace **de::vfh::portal** einordnen sollen.

Oben ist dann ein Präsentationspaket zu sehen. In diesem befindet sich ein Paket für einen Rich-Client, wie z. B. für ein Swing GUI und einen Thin-Client, wie z. B. eine JSP-Anwendung.

In der Mitte ist das Applikationspaket zu sehen, das vier Unterpakete enthält. Einen Front-Handler für die Kommunikation mit den GUIs, einen Controller für die Ablaufsteuerung, die Domain-Objects (also die Business Objekte) und ein Paket für das Management mit externen Applikationen.

Unten sehen Sie verschiedene Layer für die Datenbankankbindung.

Das Paketdiagramm zeigt also die hierarchische Struktur der Modellelemente und lässt dabei Schichten und Komponentenblöcke erkennen.



Beispiel

Abb.: Paketdiagramm (zum Vergrößern klicken)

In der Notation gibt es noch Paket Merges, die hier nicht weiter erläutert werden und in der Praxis nicht so sehr wichtig sind. Versuchen Sie selbst dazu etwas zu finden.



Hintergrundwissen

Warum sind Paketdiagramme so wichtig?

- In der Praxis sind Paketdiagramme wichtig, weil die Entwickler bereits frühzeitig nach einer „Spielwiese“ fragen und schon Testcode entwickeln wollen.
- Pakete können wie Komponentendiagramme, die wirkliche Architektur des Systems widerspiegeln. Design Patterns und Schichten in der Anwendung werden hier transparent. Wichtig ist, dass die Transparenz frühzeitig gegeben ist, von vornerein ausreichend darüber diskutiert wird oder erfahrene Designer zum Einsatz kommen.

Am wichtigsten sind Paketdiagramme, weil man mit ihnen die Abhängigkeiten modellieren kann. Versuchen Sie also das Paket zu zeichnen und sich dann zu fragen: Wer benutzt wen? Zeichnen Sie das mit Pfeilen in das Diagramm hinein. Dies zeigt Ihnen die „Benutzt-Struktur“. Diese Benutzt-Struktur ist für ein gutes Design bzw. eine gute Architektur extrem wichtig.

Liegt hier eine völlig wirre Struktur vor, wer wen benutzt? Oder gibt es einen klaren Fluss von oben nach unten? Letzteres ist für eine änderbare Architektur sehr wichtig.

Beispielsweise kann man in einem Paketdiagramm, welches benutzte Beziehungen enthält, auch sehr schön erkennen, ob es Zyklen in der Verwendung gibt. Also verwendet Paket A das Paket B und umgekehrt? Wenn dies der Fall sein sollte, ist oftmals ein Redesign vonnöten.

Im letzten Kapitel sehen wir, dass es viele Werkzeuge gibt, die die Benutztrelation zwischen

Paketen automatisch analysieren und auf Probleme hinweisen.

Aber erleben Sie das jetzt schon einmal selbst, indem sie ein Paketdiagramm so gestalten, dass neben dem hierarchischen Namensraum auch eine hierarchische Benutzrelation vorliegt. Ihr Code wird wartbarer und unempfindlicher bei Änderungen.

4.4 Klassendiagramm

Ein Klassendiagramm visualisiert Klassen auf verschiedene Art und Weise. Dazu werden einmal die Klassen selbst dargestellt, ggf. auch deren Beziehungen zueinander und evtl. auch deren innere Struktur. Üblich sind zwei Ziele:

- Darstellung der Klassen und Beziehungen: Hierbei werden verschiedene Klassen eines Paketes oder einer Komponente im Diagramm dargestellt. Dabei ist es wichtig zu zeigen, welche Klassen es überhaupt gibt und welche Klasse von anderen verwendet wird. Letzteres nennt man Assoziationen.
- Darstellung der inneren Struktur: Der Autor des Klassendiagrammes will dann zeigen oder definieren, welche Attribute oder Methoden die Klassen haben.

Beide Formen werden oft miteinander kombiniert und bilden dann ein ausführliches Klassendiagramm. Es können aber noch mehr Informationen im Klassendiagramm untergebracht werden:

1. Welchem **Stereotyp** entspricht die Klasse? Beispielsweise eine Darstellungsklasse, eine Controllerklasse, etc. Dies wird später näher erläutert.
2. Von welcher Struktur sind die Attribute und Methoden? Welche Signaturen haben die Methoden? Hierbei geht es also um Sichtbarkeiten, wie **public** / **private**, die Typen, wie **int** oder **String** oder die Parameter, wie **int b**, **String c**.



Beispiel

Einfaches Klassendiagramm

Das folgende Beispiel zeigt ein einfaches Klassendiagramm, welches zwar Attribute und Methoden mit ihren Sichtbarkeiten (+ und -) zeigt, aber noch keine umfangreicheren Assoziationen.

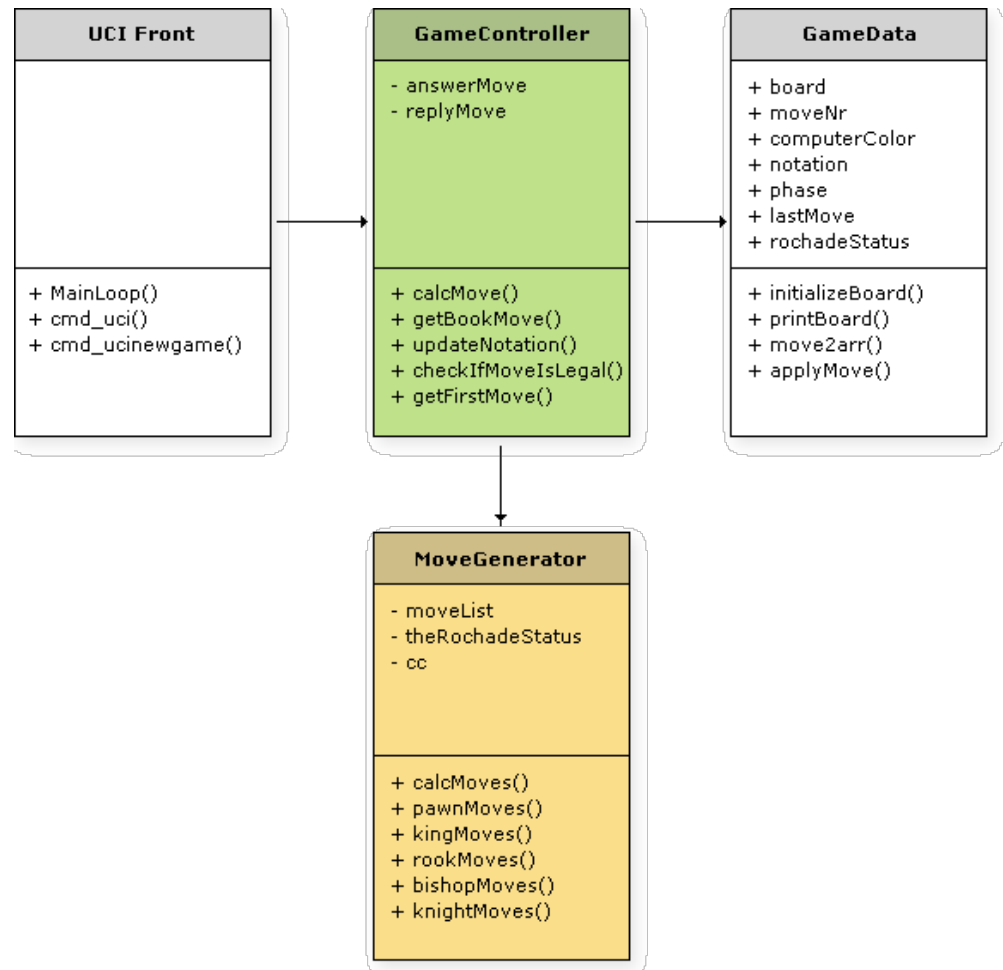


Abb.: Beispiel Klassendiagramm

Bei der UML 1 hat man noch strikt zwischen einer **Operation** und einer **Methode** - als Implementierung einer Operation unterschieden. Dies ist aber seit UML 2 nicht mehr der Fall.

4.4.1 Notation

Die Klasse selbst wird in einem Rechteck dargestellt, welches 1-3 Bereiche enthält. Oben steht der Klassenname, der meist in fester Schrift dargestellt wird. Darunter folgen erst die Attribute und dann die Methoden. Klassennamen werden in fast allen Sprachen groß geschrieben.



Hinweis

Es ist sinnvoll, die Verantwortlichkeit von Klassen als Kommentar anzugeben, also was die Klasse selbst leisten soll, sofern dies nicht trivial aus der Klasse selbst hervorgeht.

Es folgen die Attribute, die mit oder ohne Parameter oder Sichtbarkeit angegeben werden können. Auch hier sind Sichtbarkeiten, Typen, Parameter, Zusicherungen (Constraints) oder Initialwerte optional.

Klassenattribute werden genauso dargestellt und gehören einmalig zu jeder Klasseninstanz. D. h. alle Instanzen können auf dieses Attribut zugreifen.



Abb.: Klasse im Klassendiagramm

Abstrakte Klassen können sowohl kursiv, als auch fett gekennzeichnet/ausgezeichnet werden. Meistens wird **{abstract}** noch unter den Klassennamen geschrieben.

Als **Sichtbarkeiten** gibt es üblicherweise public +, protected #, private - und package ~! Diese werden mit den angegebenen Symbolen voran notiert.

Abgeleitete Attribute können in der UML mit einem Schrägstrich „/“ dargestellt werden.

Interessant ist, dass - anders als in den meisten Programmiersprachen - zuerst der Name und dann der Typ dargestellt wird. Hier also nicht verwirren lassen. Ein Beispiel:

```
person: String = "Katharina"
```

Dies gilt ebenfalls für die Parameter einer Operation / Methode! Also ist

```
getMwst (betrag: Integer)
```

eine gültige Angabe für eine Methode.

4.4.2 Interfaces

Als weitere Auszeichnungen für Klassen gibt es z. B. **Enumerations** oder **Interfaces**. Enumerations werden mit `<<enumeration>>` notiert. Im Body der Enumeration sind dann Enumerationsmengen (z. B. rot, grün, blau, etc.) zu finden.

Viel wichtiger sind aber **Interfaces**, d. h. Schnittstellen, die einen zu implementierenden **Kontrakt** in Form von Methodenangaben vorgeben. Sie werden mit dem Schlüsselwort `<<interface>>` über dem Klassennamen notiert.

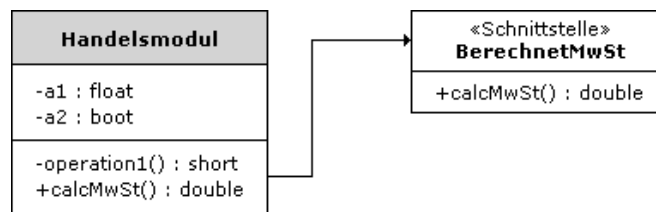


Abb.: Beispiel Interface

Man sagt hier, das Handelsmodul **realisiert** oder **implementiert** die Schnittstelle, die die Mehrwertsteuer berechnet.

Attribute werden nicht in Interfaces notiert (ist aber auch nicht verboten), da es um einen Zugriffsvertrag geht und nicht um die innere Struktur der Klasse.

Da Interfaces strikte Verträge sind, findet man hier weitergehende Spezifikationen wie Ausnahmen, Invarianten sowie Vor- und Nachbedingungen.

Eine weitere Möglichkeit, Schnittstellen zu notieren ist die Darstellung als Kugel und Halbkreis. Die Kugel implementiert die Schnittstelle und der Halbkreis ist das Interface, an das quasi „angedockt“ wird.



Abb.: Interface als Kugel und Halbkreis

Zusätzlich kann man mit einem Wort den Halbkreis beschreiben, welche Eigenschaft diese Klasse dann implementiert (z. B. serialisierbar, oder sortierbar).

4.4.3 Stereotypen



Definition

Stereotyp

Ein Stereotyp ist ein gleichbleibendes Muster, das auftritt oder wiederverwendet werden kann. Der Begriff stammt eigentlich aus der Psychologie und der Soziologie und bezeichnet gleiche Verhaltensweisen von Menschen oder Gruppen.

Auf die UML und Klassen übertragen bedeutet dies, dass man eine Klasse oder andere UML-Elemente mit einem Stereotyp auszeichnen / klassifizieren kann. Sie ermöglichen dadurch die Verwendung von plattform- oder domänenspezifischer Notation, wodurch sie gewissermaßen markiert wird, ein bestimmtes Verhalten zu implementieren. Zum Beispiel **<<Stateless>>** für eine **Stateless Session Bean**.

Seit der UML 2 können Klassen mehrere Stereotypen beinhalten. Die meisten UML Werkzeuge erlauben es, Stereotypen direkt den Elementen wie Klassen, Abhängigkeiten, Komponenten oder Pakete zuzuweisen. Stereotypen werden dann auch oft in gleichen Farben dargestellt.

Die UML gibt bereits eine Reihe so genannter „Standard-Stereotype“ vor, die Ihnen für Ihre Modellierung zur Verfügung stehen. Nachfolgend finden Sie einige der relevanten Standard-Stereotypen:

- **<<focus>>** sagt aus, dass die so gekennzeichneten Klassen die zentrale Logik oder den zentralen Kontrollfluss definieren.
- **<<auxiliary>>** implementieren üblicherweise sekundäre Logik oder sekundären Kontrollfluss und wirken dadurch unterstützend für andere Klassen, die für das Modell zentral sind.
- **<<utility>>** definieren eine Sammlung von statischen Attributen und Klassen-Operationen, auf die von allen Klassen aus zugegriffen werden kann.
- **<<create>>** kann einer von zwei vordefinierte Stereotypen sein, von denen sich einer auf Abhängigkeitsbeziehungen und einer auf Operationen anwenden lässt.
- **<<destroy>>** ist invers zu create und besagt, dass die Operation eine Ausprägung des Classifiers, in dem sie enthalten ist, zerstört.
- **<<call>>** gibt an, dass eine so markierte Abhängigkeitsbeziehung an ihren Enden jeweils Operationen oder Klassen mit Operationen besitzt, auf welche sich die Assoziation bezieht.
- **<<instantiate>>** bedeutet angebracht an eine Abhängigkeitsbeziehung, dass Operationen des Verbrauchers Ausprägungen des Anbieters erzeugen.
- **<<interface>>** ist eine Schnittstelle. Von ihr können keine Ausprägungen erzeugt werden, sie enthält keine Attribute, und ihre Operationen können nicht in Methoden implementiert werden.
- **<<send>>** wird auf Abhängigkeitsbeziehungen angewandt und besagt, dass die Quelle der Beziehung (eine Operation) ein Signal verschickt.
- **<<type>>** wird an eine Klasse angebracht und sagt so aus, dass diese Klasse eine Menge von Objekten hinsichtlich der Operationen, die auf diese Objekte angewendet werden können, gruppiert.
- **<<responsibility>>** impliziert eine Verpflichtung oder einen Vertrag zwischen zwei Modellelementen.

4.4.4 Assoziation

Die Assoziation beschreibt eine Beziehung der Klassen zueinander, bei der ein Objekt das andere verwendet.

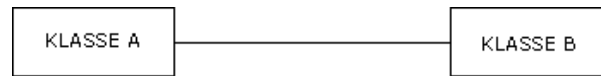


Abb.: Assoziation

Dazu ein Beispiel. Nehmen wir an, ein Kunde wäre die Klasse. Dieser Klasse können mehrere Rechnungen zugeordnet werden. Das Objekt Kunde, muss auf Rechnungsobjekte zugreifen können. Sonderfälle der Assoziation sind die Aggregation und die Komposition.

Dargestellt wird die einfache Assoziation mit einer einfachen Linie.

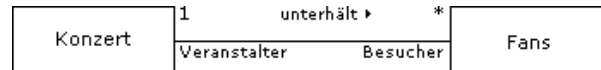


Abb.: qualifizierte Assoziation

Wichtig sind dabei die Multiplizitäten, mit denen angegeben wird in welcher Anzahl die beiden Objekte miteinander verknüpft sind. Beispiele sind: **0..1**, **0..42**, **1..1**, **1..42**, **0..***, **1..***

In E/R-Diagrammen spricht man auch oft von **1:1**, **1:N** oder **N:M** - Beziehungen. Allgemein sind Relationen also wichtige Hinweise, wenn beispielsweise Objekte, Datenbankschemata oder Datenbank-Mapping Information angegeben werden.

Weiterhin können Assoziationen qualifiziert werden, indem der Strich noch mit **Rollen** und **Eigenschaften** ausgezeichnet wird.

Und schließlich gibt es mehrgliedrige Assoziationen, bei denen mehrere Klassen verknüpft sind. In diesem Fall laufen die Linien zu einer kleinen Raute zusammen.

Gerichtete Assoziation

Die gerichtete Assoziation beschreibt, dass man über die Ausgangsklasse zur Zielklasse gelangen kann. Aber nicht umgekehrt. Es handelt sich also um eine **unidirektionale** Beziehung.

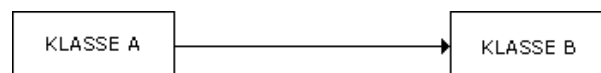


Abb.: Gerichtete Assoziation

Soll in beide Richtungen - also **bidirektional** - navigiert werden können, dann können zwei Pfeile zu einem zusammengefasst werden. Diese können dann ebenfalls mit Rollen und Eigenschaften versehen werden.

Aggregation

Eine Aggregation ist ein Spezialfall einer Assoziation. Sie wird als **Teile-Ganzes Beziehung** bezeichnet.

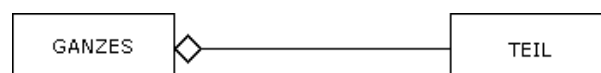


Abb.: Aggregation

Man schreibt daher auch oft „besteht aus“ als Eigenschaft an den Pfeil. Ein Beispiel wäre eine „Party“ als Ganzes, die aus vielen Teilnehmern besteht. Wird die „Party“ und damit die Aggregation aufgelöst, sind die Teilnehmer dennoch überlebensfähig. Sie brauchen die Party nicht zwingend. Jedoch ergibt sich die Party nur aus der Aggregation der Partyteilnehmer.

Wer im Zweifel ist, ob eine Aggregation vorliegt, kann eine Assoziation verwenden, ohne einen großen Fehler zu begehen. Eine deutliche Auszeichnung ist jedoch immer lesbarer und nützlicher.

Komposition

Eine Komposition wird im Gegensatz zu Aggregation mit einer ausgefüllten Raute gezeichnet.

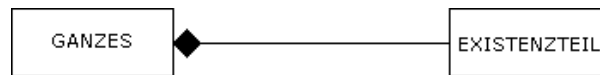


Abb.: Komposition

Sie ist die stärkste Form der Aggregation. Die Teile sind für das Ganze existenzabhängig. Das bekannteste Beispiel ist hier die Rechnung mit ihren Rechnungsposten. Wird die Rechnung gelöscht, so hören auch die Rechnungsposten auf zu existieren.

Anders bei einem Verein (Aggregation) zu seinen Mitgliedern. Bei der Löschung eines Vereines aus dem Amtsregister möchten und können die bisherigen Mitglieder durchaus noch weiterleben.

Abhängigkeit

Eine Klasse benötigt eine andere zwingend. Nur dann kann sie ihre Aufgabe erfüllen.

4.4.5 Vererbung / Generalisierung / Spezialisierung

Bei der Vererbung sind alle Attribute und Methoden auch in der Unterklasse verfügbar. Notiert wird dies in UML mit einem **Pfeil**, der **nicht ausgefüllt** ist.

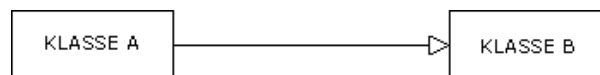


Abb.: Vererbung

Sprachlich unterscheidet man hier die **Generalisierung** und die **Spezialisierung**.

Die Generalisierung bezeichnet die allgemeineren Eigenschaften der Oberklassen.

Die Spezialisierung bezeichnet die Unterklassen, die denen der Oberklasse weitere spezielle Eigenschaften (Attribute) und Methoden hinzufügt.

In vielen Fällen ist die Delegation sinnvoller als die Vererbung, da sie das Geheimnisprinzip nicht verletzt. In vielen Fällen - wie z. B. bei Grafikbibliotheken - ist jedoch die Vererbung sinnvoll. Denken Sie daran, dies immer selbst gut zu analysieren.

In einigen Programmiersprachen ist Mehrfachvererbung erlaubt. Pfeile können in diesem Fall - oder wenn mehrere Unterklassen von einer Superklasse (Oberklasse) ableiten - zusammengefasst werden.

4.5 Kompositionsstrukturdiagramm

Oftmals reichen die vorhandenen Diagramme nicht aus, um zu zeigen aus welchen Elementen sich eine Klasse oder eine Komponente zusammensetzt.

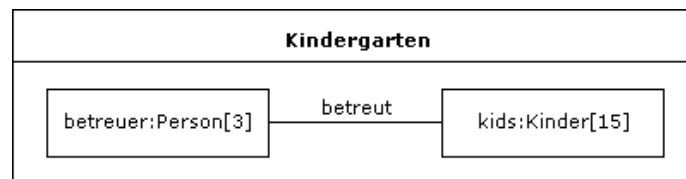


Abb.: Kompositionsstrukturdiagramm

Gezeichnet wird dieses Diagramm mit der übergeordneten Komponente in einem Kasten. Dieser enthält zwei Unterelemente. In diesen Elementen kann der Name des Parts (Teils) gefolgt von einem Doppelpunkt und einem Typ des Parts dargestellt werden.

Die **Multiplizitäten** werden dahinter entweder in eckigen Klammern oder hochgestellt (wie zum Quadrat) angegeben. Die Parts können wiederum mit einer Rolle verbunden sein.

4.6 Objektdiagramm

Das Objektdiagramm zeigt den beispielhaften Zustand eines oder mehrerer Objekte in einem System. Es ist wie eine Fotografie des Systems, in dem die Objekte vielleicht gerade besonders interessante Werte haben. Es ist quasi ein Instanzdiagramm, in der konkrete Attributwerte angegeben werden können.



Abb.: Objektdiagramm

Dieses Bild zeigt also Klassen in einem bestimmten Zustand.

Noch ein paar Punkte:

- Bei der Definition der Objektklasse oder des Attributtypes ist man sehr frei. Diese Elemente können auch einfach weggelassen werden.
- Dieses Diagramm kann auch sinnvoll sein, wenn OCL-Ausdrücke mit angegeben werden. Dies würde z. B. so geschehen: `contact Reifen inv: groesse > 20` (also sind zu kleine Reifen nicht erlaubt).
- Wie oben eingezeichnet können auch Links zwischen den Objekten eingezeichnet werden. Dies sind dann quasi die Assoziationen und werden nur mit der Rolle weiter verfeinert.

5 Dynamische Diagramme

In diesem Teil geht es mit den dynamischen Diagrammen weiter:

- Anwendungsfalldiagramm -> Welche Anwendungsfälle gibt es?
- Aktivitätsdiagramm -> Wie sieht ein Web durch die Anwendung aus?
- Zustandsdiagramm -> Welche Zustände kann das Programm annehmen?
- Sequenzdiagramm -> Wie kommunizieren Klassen / Objekte miteinander?
- Interaktionsübersicht -> Ein Aktivitätsdiagramm welches einzelne Aktivitäten durch Interaktionen ersetzt.
- Kommunikationsdiagramm -> Stellt komplexe Kommunikationsabläufe dar
- Zeitdiagramm -> Zustandslinien für den Embedded- oder den Elektronikbereich

5.1 Anwendungsfalldiagramm

Anwendungsfalldiagramme werden sowohl im Englischen als auch im Deutschen als **Use-Case Diagramme** bezeichnet. Dieses Diagramm ist recht wichtig und ist fast das erste UML-Diagramm, welches in Projekten gezeichnet wird.

Ziel ist es möglichst einfach zu zeigen, was man mit dem zu bauenden Softwaresystem machen will. Welche Fälle der Anwendung es also gibt! Aus der Analysephase weiß man, dass es Anwendungsfälle zu dokumentieren gilt. Darin werden Arbeitsabläufe im System beschrieben, die möglich sein sollen. In der UML soll dies nun noch einfacher mit simplen Grafiken unterstützt werden.

Das Diagramm zeigt sogenannte **Akteure** als Strichmännchen. Dies können beispielsweise Personen wie Kunden, Administratoren oder beliebige andere Anwender sein.

Von ihnen aus gehen Striche zu den jeweiligen Anwendungsfällen, die in eine Ellipse hineingeschrieben werden. Um zu gruppieren notiert man zusammenhängende Anwendungsfälle in einen Kasten. Dazu ein sehr einfaches Beispiel:

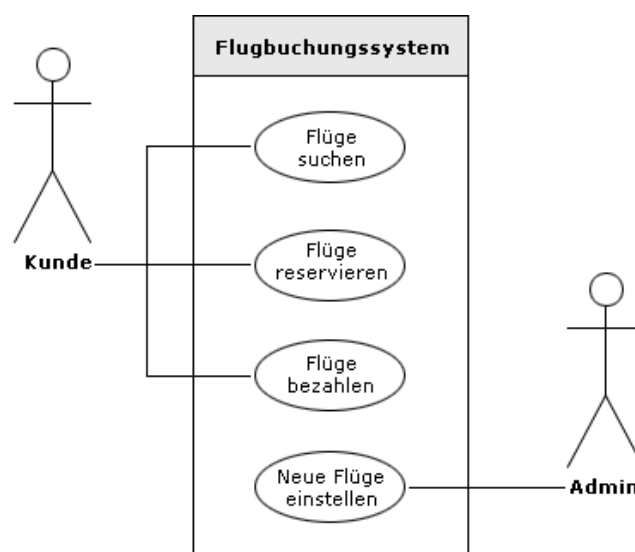


Abb.: Anwendungsfalldiagramm

Mit dem Anwendungsfalldiagramm sollen die Anwendungsfälle möglichst einfach kommuniziert werden - beispielsweise dem Management. Es genügt daher meist schon ein Blick auf ein solches Diagramm, und der Betrachter kann sofort selbst eine Vision vom System entwickeln oder eigene Ideen vorschlagen. Dies ist oft effizienter als das Lesen von umfangreicher Anwendungsfalldokumentation.

Die Striche zwischen den Akteuren und Anwendungsfällen können wie beim Klassendiagramm benannt und mit Kardinalitäten versehen werden. So das z. B. immer nur zwei Piloten den Anwendungsfall „Autopilot abschalten“ programmiert haben möchten.

Man beachte in obigem Beispiel das man das Gefühl hat, die Aktionen finden von oben nach unten statt. Aber **Anwendungsfalldiagramme sind nicht gerichtet!** Für den zeitlichen Ablauf sind Aktivitätsdiagramme da. Die Diagrammart zu vermischen ist ein typischer Fehler von UML - Einsteigern. Komplexe Anwendungsfälle in einem Use-Case Diagramm mit vielen Pfeilen zu verbinden ist nicht korrekt.

Weitere Anwendungsfallbeziehungen

1. Mit einem gestrichelten Pfeil wird dargestellt, dass eine Realisierung vorliegt. D. h. der Anwendungsfall „Zur Meldestelle gehen“ kann beispielsweise dadurch realisiert werden, dass es zwei mit gestricheltem Pfeil verbundene Fälle gibt, die z. B. „mit dem Auto zur Meldestelle fahren“ und „zu Fuß zur Meldestelle gehen“ heißen kann. Beide zusammen realisieren den Hauptanwendungsfall.
2. Schreibt man an den gestrichelten Pfeil `<<include>>`, so wird der Anwendungsfall auf den gezeigt wird, mit eingebunden. Quasi so, wie man das von Programmiersprachen kennt. Besteht also der Anwendungsfall intern aus drei Teilen A, B, C, so kann man Teil A als Anwendungsfall auslagern und mit `<<include>>` und gestricheltem Pfeil verbinden und wenn es wichtig ist, diesen nochmal extra darstellen.
3. Mit `<<extends>>` und einem gestrichelten Pfeil sagt man dann umgekehrt, dass ein Anwendungsfall mit einem weiteren erweitert werden kann. Die Pfeile gehen dabei immer vom **extend**- / **include**-Objekt zum Hauptobjekt, welches diese nutzt.
4. Mit einem **Spezialisierungspfeil** (analog zu dem im Klassendiagramm) zeigt man an, dass mehrere „Subanwendungsfälle“ einen generelleren Anwendungsfall ausmachen. So kann ein Use-Case beispielsweise die „Standard-Authentifizierung“ sein, die entweder mit einer „Elektronischer-Personalausweis-Authentifizierung“ oder mit einer „Handy-Nummer-Authentifizierung“ durchgeführt wird.

5.2 Aktivitätsdiagramm

Das Aktivitätsdiagramm wird im Englischen als **Activity Diagram** bezeichnet. Es ist mit seinen vielen Elementen eines der komplexesten Diagramme. Hier können nicht alle Elemente wiedergegeben werden, diese sind aber leicht in der UML-Übersicht von oose nachzuschlagen. Wir beschränken uns hier auf die wichtigsten Elemente.

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

Das Aktivitätsdiagramm wird meist recht früh in der Modellierungsphase angewendet und zeigt die Abfolge, in der der Anwender durch einen möglichen Weg der Anwendung geleitet wird. Dazu gibt es **Aktionsknoten**, **Objekt-** und **Kontrollknoten**. Letztere erlauben es zu verzweigen und Entscheidungen zu treffen.

Aktivitätsdiagramme visualisieren und erklären den Ablauf des Programmes, der oft mit der natürlichen Sprache oder dem geschriebenen Wort nur sehr schwer dargestellt werden kann.

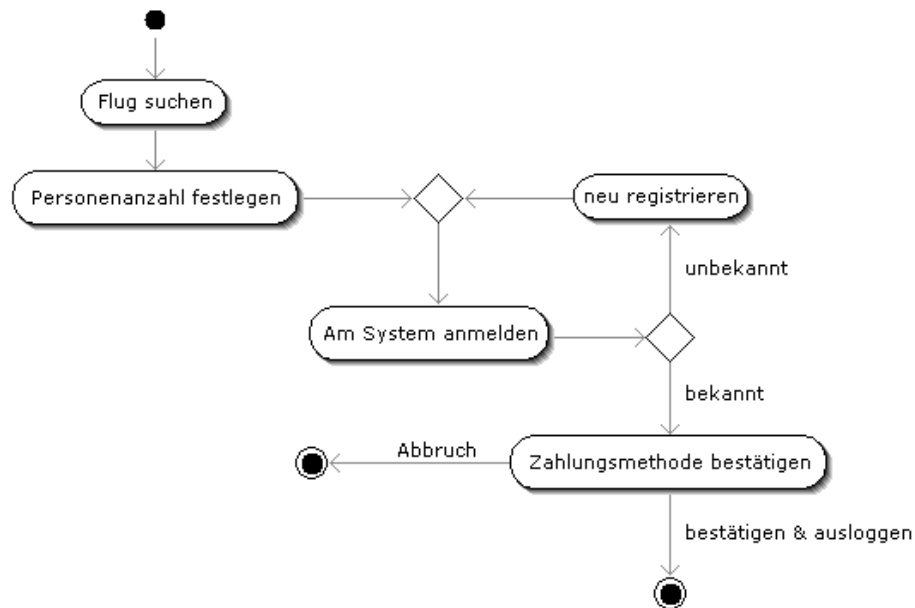


Abb.: Aktivitätsdiagramm

In diesem Diagramm sind einige Aktionsknoten (zu Zeiten von UML 1 noch Aktivitäten genannt) zu sehen. Desweiteren ein Startknoten und ein Endknoten. Rechts ist eine Verzweigung zu sehen. Nicht eingezeichnet sind Objektknoten. So könnte man an diversen Stellen Objekte einfügen, die an dieser Stelle dann vorhanden sind.

Beispielsweise ein Flugobjekt, welches nach der erfolgreichen Suche vom Kunden ausgewählt wird. Oder ein Objekt welches den gesamten Flug einschließlich der Kunden- und Zahlungsdaten repräsentiert.

Wie in vielen UML 2 - Diagrammen ist es möglich, Diagramme in einen Kasten zu packen und damit zu veranschaulichen, dass eine Aktivität aus mehreren Teilschritten bestehen kann.

Kontrollknoten

Verzweigungen, Zusammenführungen oder Entscheidungen werden mit Kontrollknoten dargestellt. Bei Zusammenführungen können dabei Bedingungen angegeben werden und bei Verzweigungen können Bedingungen an die ausgehenden Arme geschrieben werden.

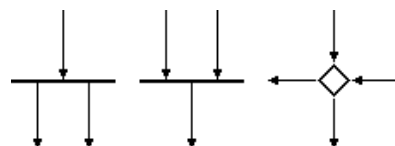


Abb.: Kontrollknoten

Für zusammenführende Knoten können Bedingungen so formuliert werden:

$\{joinSpec=(X \text{ or } Y) \text{ and } (M \text{ or } N)\}$.

Für Verzweigungen können Bedingungen in rechteckigen Klammern angegeben werden: $[X<0]$.

Weiterführendes

1. **Parameter Pins für Objektknoten:** Pins stellen In und Out Parameter für Objekte dar.
2. **Schwimmbahnen:** (engl. Swimlanes) können Bereiche darstellen, wie das in Sequenzdiagrammen der Fall sein kann.
3. **Signale:** In den Kontrollfluss vieler Diagramme können Signale eingebettet werden. Diese werden, wie unten stehend, dargestellt.

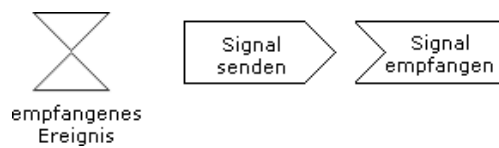


Abb.: Signale

5.3 Zustandsdiagramm

Das Zustandsdiagramm beschreibt das Verhalten eines Systems bei bestimmten Ereignissen. Ein solcher Zustandsautomat geht auf alte Modellierungsversuche von Mealy- und Moore-Automaten zurück.

Er enthält:

- Zustände (hierzu gehören auch zusammengesetzte Zustände und ganze Unterzustände)
- Transitionen
- Regionen / Bereiche
- Start- und Endzustand

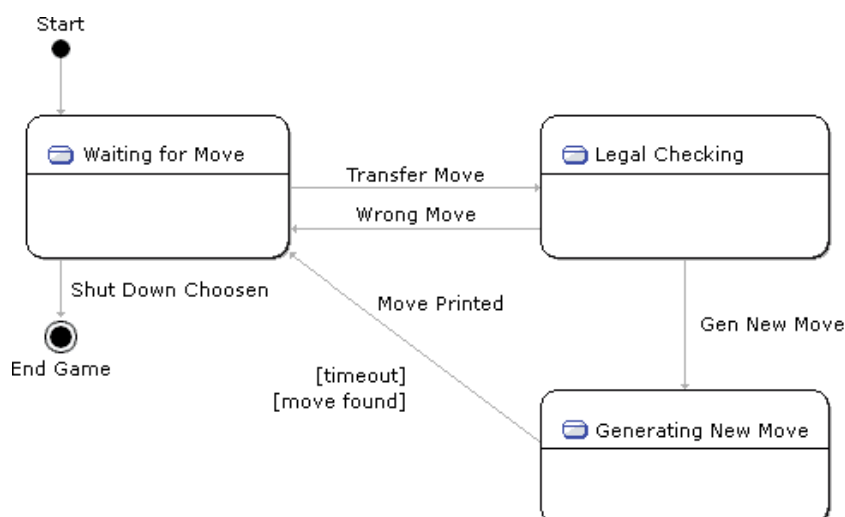


Abb.: Zustandsdiagramm

Die Notationsmöglichkeiten unterscheiden sich doch stark von Anwendung zu Anwendung. Im einfachsten Falle liegt pro Zustand nur ein Kasten vor, in dem der Zustand beschrieben ist. In diesem Falle sollte (anders als im Bild hier) die waagerechte Linie wegfallen. Dargestellt sind hier die drei einfachsten Zustände eines (Schach-)Programmes, welches drei Zustände einnehmen kann.

Üblicherweise wird jedes Diagramm mit einem Kasten umrahmt, in welchem oben der Name des ganzen Zustandsdiagrammes steht.

Weiterhin ist es möglich, Anwendungsfälle im Use-Case Diagramm durch Zustandsautomaten zu verfeinern. Dies wird z. B. oft bei der Authentifizierung gemacht.

Notationselemente

Ein Zustand wird meistens (aber optional) mit einem verständlichen Namen im oberen Kasten versehen.

Über Zustände können in dem Zustand die folgenden Angaben gemacht werden:

1. **entry** / Aktivität -> Eintrittsaktivität
2. **exit** / Aktivität -> Austrittsaktivität
3. **do** / Aktivität -> anhaltende Aktivität
4. Trigger / defer -> Verzögerung, dabei unterscheidet man:
 - SignalTrigger
 - CallTrigger
 - TimeTrigger
 - ChangeTrigger und
 - AnyTrigger für alles

Ein **Guard** ist eine Bedingung, die an einer Transition angegeben werden kann.

Weitere Elemente

Weitere Elemente sind bereits aus anderen Diagrammen (wie Aktivität) bekannt.

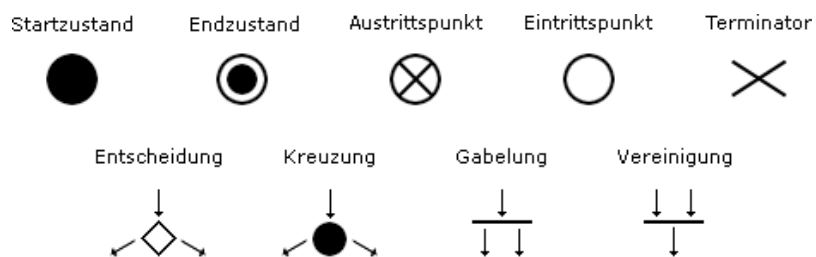


Abb.: Diagrammelemente

5.4 Sequenzdiagramm

Sequenzdiagramme beschreiben eine dynamische Sichtweise auf das System. Man verwendet es also, wenn man zeigen will wie Klassen interagieren.

Da man bei Methodenaufrufen zwischen Klassen auch von Nachrichten spricht, die versendet werden, zeigt das Diagramm die Nachrichten im zeitlichen Ablauf.

Oben angeordnet werden dazu (unterstrichene) Objekte - es sind aber auch Klassen oder sogar noch größere Einheiten möglich - gezeigt. Sie haben Linien nach unten (sogenannte **Schwimmbahnen** (engl. swimlanes)) oder auch **Lebenslinien** (engl. lifelines).

Auf diesen Linien sieht man in Blöcken, wie Objekte instanziiert werden. Von ihnen gehen dann **Nachrichten** an andere Objekt(blöcke). Diese werden als **vertikale Pfeile** dargestellt. Nachrichten können Rückpfeile (return arrows) haben. Die Diagramme erlauben die Darstellung von synchronen (gefüllte Pfeile) und asynchronen Nachrichten (nicht gefüllte Pfeile).

Gleichzeitig können wie in den vorigen Diagrammen Signale empfangen werden oder auch Verzweigungen, Iterationen und Rekursion dargestellt werden. Dazu werden Kästen in das Diagramm gemalt und dann beispielsweise **loop** (Schleifen) oder **alt** (if) eingezeichnet.

Eine initiale Startnachricht (engl. found message) kann das Sequenzdiagramm starten.

Das nachfolgende Diagramm von  **SUN** zeigt einen der besten und typischsten Anwendungsfälle für Sequenzdiagramme - **Design Patterns**. Gezeigt wird hier eine gute Architektur für den Datenzugriff.

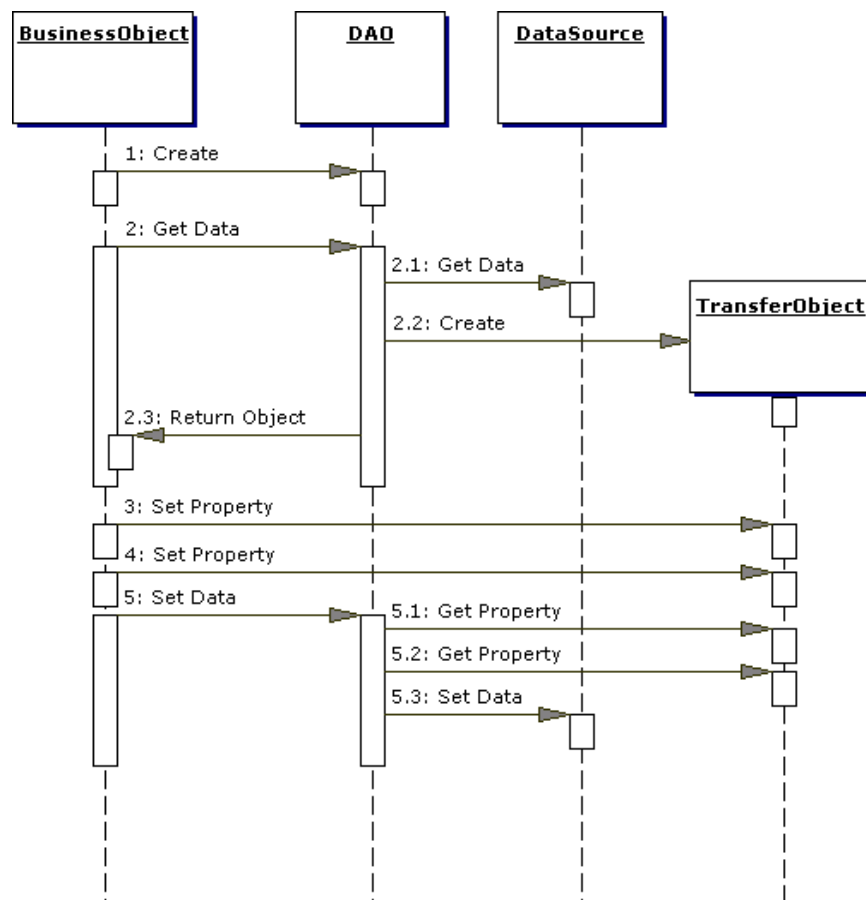


Abb.: Sequenzdiagramm

Was passiert hier? Gezeigt ist das **DAO**-Pattern. Es beinhaltet vier Objekte. Gehen wir es einmal kurz durch. Nicht um das Pattern zu verstehen, sondern um zu zeigen, weshalb Sequenzdiagramme nützlich sind.

1. Ein **Business Objekt** benötigt Daten aus der Persistenzschicht (der Datenzugriffsschicht, also letztlich aus der Datenbank).
2. Es wendet sich an ein DAO Objekt das für alle Datenbankverbindungen zuständig ist.
3. Dieses DAO weiß, welcher Datenzugriff gerade aktuell ist und holt sich die Datenbankverbindung über die **DataSource**.
4. Ein **Data Transfer Objekt** enthält dann die Daten in einer passenden Form (z. B. besser als ResultSets) und dieses wird in 2.3 an das Business Objekt zurück gegeben.

Was danach von 3 bis 5.3 folgt ist die Rückrichtung, die uns hier inhaltlich nicht besonders interessieren muss. Der Form halber: Anstatt Daten z. B. per **JDBC** zurückzuschreiben ist es heute State-of-the-Art, das Transfer Objekt zu verändern und es dann über das DAO speichern zu lassen.

Sie haben anhand des Szenarios gesehen, wie hilfreich Sequenzdiagramme sind. Der Sachverhalt der Kommunikation von vier wichtigen Objekten in der Datenhaltungsschicht wird durch die Darstellung transparenter. Zusammen mit dem **Active Record Design Pattern** zeigt das dargestellte Beispiel die wichtigsten Design Patterns der Datenhaltungsschicht.

Interessant ist, dass es in Sequenzdiagrammen oft zwei Arten von Kommunikationsschemas gibt:

1. **Centralized Control:** Ein Objekt macht die Arbeit (rechnet oder hält die Fäden wie eine Spinne in der Hand) und alle anderen arbeiten zu.
2. **Distributed Control:** Alle Klassen / Objekte sind gleichberechtigt und „rechnen“ quasi auch. Es gibt keine zentrale Kontrolle:

In vielen Fällen ist die zweite Struktur vorteilhafter und ermöglicht mehr objektorientierte Konzepte (Polymorphie, Überschreiben etc.).

Zur Vertiefung schauen Sie sich das folgende Tutorial an:

 [Tutorial: Sequenzdiagramme \(Siehe Anhang\)](#)

5.5 Interaktionsdiagramm

Ein Interaktionsdiagramm ist eine Mischform eines Aktivitätsdiagrammes in welchem einzelne Aktivitäten näher durch ein Interaktionsdiagramm erläutert werden.

Konkret liegt also ein Aktivitätsdiagramm vor, bei dem einzelne Zustände näher erläutert werden sollen. Dann werden diese Zustände im Diagramm vergrößert und ein Interaktionsdiagramm eingebettet d. h. angezeigt. Dies wird immer dann genutzt wenn Zustände in Aktivitätsdiagrammen sehr komplex sind.

5.6 Kommunikationsdiagramm

Das Kommunikationsdiagramm stellt komplexere Abläufe der Kommunikation dar.

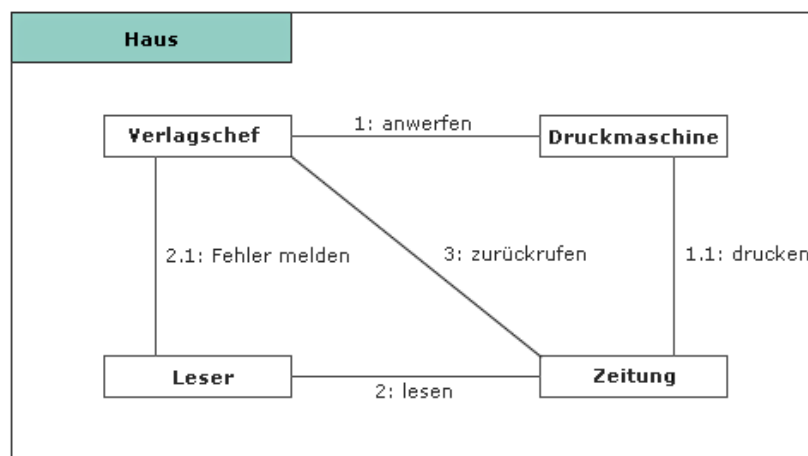


Abb.:
Kommunikationsdiagramm

Was wir hier sehen ist der Versuch, einen komplexen Ablauf in der Reihenfolge richtig darzustellen.

- Der erste Kommunikationsstrang ist das Anwerfen und Drucken der Zeitung durch den Verlagschef.
- Der Käufer liest die Zeitung und meldet einen großen Fehler.
- Der Verlagschef zieht daraus die Zeitung wieder aus dem Verkehr.

In der Informatik wird ein solches Diagramm immer dann verwendet, wenn die Inhalte der Zustände, Übergänge und Daten nicht wichtig sind, die Elemente aber sehr wohl in einem Diagramm dargestellt werden sollen. Das heißt beispielsweise, dass die Aufgaben oder Daten der Druckmaschine nicht wichtig sind, sondern nur, dass es sie gibt und wie sie interagiert.

5.7 Zeitdiagramm

Zeitdiagramme werden meistens für den embedded Bereich oder für elektrotechnische Anwendungen verwendet. Nur in diesen Bereichen gibt es so präzise Prozesse oder Vorgänge, dass man diese zeitlich darstellen muss. In der sonst üblichen Modellierung von softwaretechnischen Vorgängen - wie z. B. ein Aktivitätsdiagramm - ist die zeitliche Abfolge egal. Ein Buchungsvorgang kann dabei Sekunden oder auch Stunden dauern.

Die Diagramme sehen wie Signale auf einem Oszilloskop aus und enthalten:

- Zeitverlaufslinien,
- Bedingungen und
- Nachrichten, die zu einer Zustandsänderung führen können.

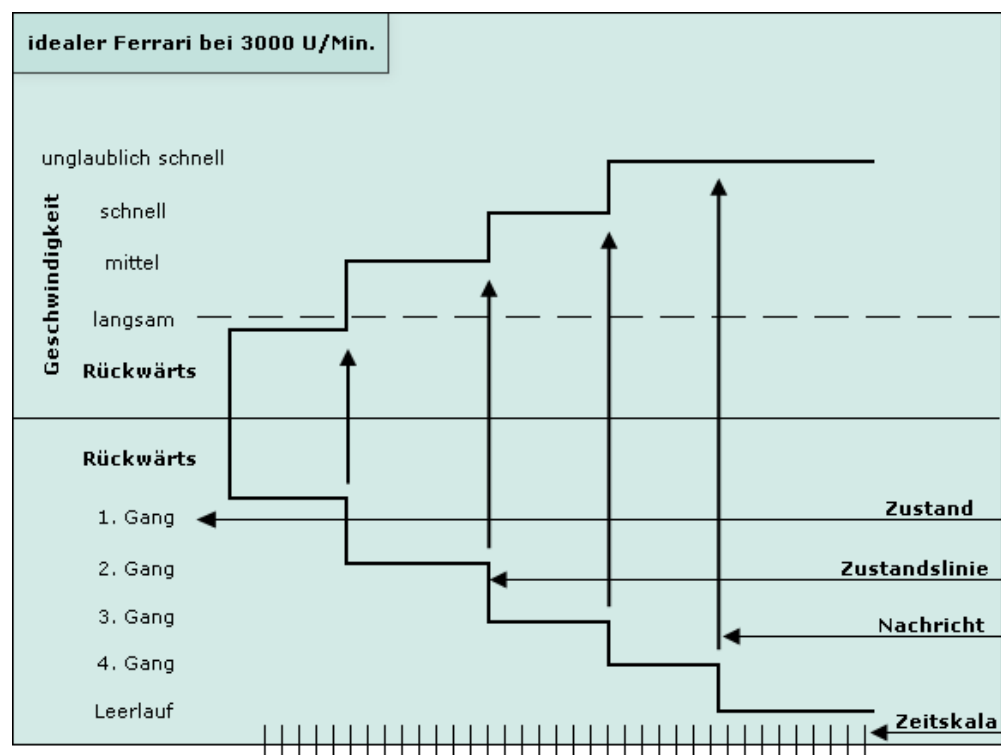


Abb.: Zeitdiagramm

Zusammenfassung

- Als eine Menge von Interaktionselementen dient die UML als ein Verfahren, mit dessen Hilfe Modelle für Softwaresysteme entwickelt werden.
 - UML hilft, den Designablauf zu bestimmen.
 - Die Väter der UML sind **GRADY BOOCH**, **JAMES RUMBAUGH** und **IVAR JACOBSEN**. Sie werden auch „Die drei Amigos“ genannt.
 - UML ist kein Dogma und nicht zwingend notwendig für die Entwicklung eines Softwaresystems.
 - Sinn der UML ist es, Ideen zu kommunizieren. Durch den Einsatz verschiedener Diagramme lassen sich Sachverhalte darstellen, die schriftlich oder sprachlich schwer darstellbar wären
 - In der UML kommen verschiedene Diagrammartentypen zur Darstellung zum Einsatz. Hier gibt es eine Unterteilung in:
 - **statische Strukturdiagramme**
(Zur Darstellung der Struktur von Modulen eines Softwaresystems. Die UML kennt sechs Strukturdiagramme: Klassendiagramm, Kompositionsstrukturdiagramm, Komponentendiagramm, Verteilungsdiagramm, Objektdiagramm und Paketdiagramm.)
 - **dynamische Diagramme**, auch Verhaltensdiagramme
(Zur Darstellung von Funktionsabläufen. In der UML kommen sieben Verhaltensdiagramme zum Einsatz: Aktivitätsdiagramm, Anwendungsfalldiagramm (auch: Use-Case o. Nutzfalldiagramm genannt), Interaktionsübersichtsdiagramm, Kommunikationsdiagramm, Sequenzdiagramm, Zeitverlaufsdiagramm und Zustandsdiagramm.)
 - Es gibt eine Vielzahl von Werkzeugen, welche das Modellieren mit UML unterstützen.
-

Wissensüberprüfung



Lückentext

Übung UML-01

Die UML ist eine Menge von Notationselementen, mit denen _____ für Softwaresysteme entwickelt werden können.

Die drei Entwickler Rumbaugh, Booch und Jacobsen werden auch die drei _____ genannt. Ziel von UML - _____ oder Beschreibungen ist, dass andere Personen die Aussage verstehen.

Kritiker allerdings sagen, die UML sei zu groß, zu komplex und _____

Ein oft nicht dokumentierter Anwendungsbereich der UML ist das _____. Hierbei haben sie vielleicht schon ein System, das weiterentwickelt werden soll und sie möchten sich einen Überblick über das System verschaffen.

Die Darstellung in der UML erfolgt mit Hilfe von Diagrammen. Paketdiagramme, Klassendiagramme, Objektdiagramme u.a. gehören zur Gruppe der _____.

Diagramme wie Zustandsdiagramm, Sequenzdiagramm, Zeitdiagramm sind _____ Diagramme.

Ein _____, zum Beispiel, zeigt, wie der Ablauf eines Programmes ist, indem Übergänge (_____) und Verzweigungen verwendet werden.

Aktivitätsdiagramm

Amigo

Diagramm

dynamisch

flexibel

Komponentendiagramm

Modell

Muchachos

redundant

Reverse-Engineering

Strukturdiagramm

Transition

? Test wiederholen Test auswerten Lösung anzeigen



Selbststudium

Übung UML-02

UML - Spezifikation

Laden Sie sich die UML Spezifikation von der OMG herunter und lesen / stöbern Sie ein bisschen darin.


Bearbeitungsdauer: 20 Minuten



Selbststudium

Übung UML-03

UML - Werkzeug

Stellen Sie sicher, dass Sie ein UML-Werkzeug „drauf“ haben. Evaluieren Sie einige der in der Lerneinheit genannten  Werkzeuge. Planen Sie dafür etwas Zeit ein. Oftmals ist man mit dem ersten Werkzeug nicht zufrieden. Beispielsweise gefällt einem der generierte Code nicht oder es fehlen einem Diagramme. Wenn ihnen dann ein Werkzeug passt, üben Sie damit und stellen Sie sicher, dass Sie sich darin fit fühlen.

Bearbeitungsdauer: 60 Minuten

Einsendeaufgaben

Bevor Sie mit der Bearbeitung der Einsendeaufgabe beginnen, besprechen Sie bitte mit Ihrer Modulbetreuung in welcher Form die Aufgaben bearbeitet werden sollen.



Einsendeaufgabe

Einsendeaufgabe UML-E1

Design eines mittelgroßen Projektes

Designen Sie ein mittelgroßes Projekt. Beispielsweise eine Mitfahrzentrale. Legen Sie sich eine Reihenfolge zurecht, welches Diagramm Sie in welcher Reihenfolge zeichnen. Sprechen Sie den Zwischenstand mit dem Betreuer durch. Senden Sie das Endprodukt an den Dozenten, der dazu Feedback geben wird.

Bearbeitungsdauer: 40 Minuten



Einsendeaufgabe

Einsendeaufgabe UML-E2

Modellierung eines „großen“ Projektes

Modellieren Sie ein großes Projekt! Entwerfen Sie beispielsweise Toll-Collect NG/2. Und nutzen Sie alle Diagramme (bis auf vielleicht das Timing-Diagramm).

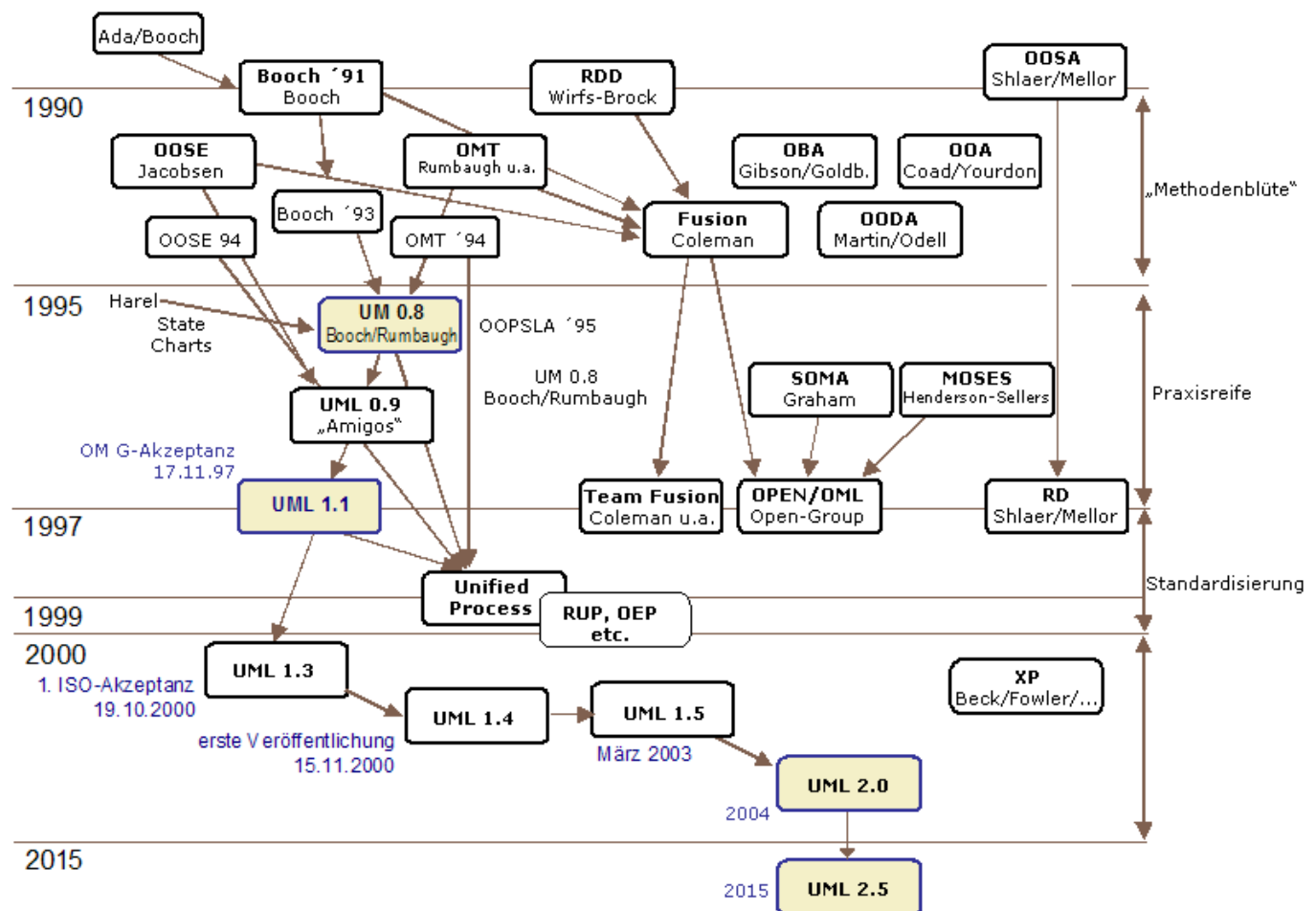
Nützliche Links:

- ReferenceCard1: <http://www.digilife.be>
- ReferenceCard2: <http://www.digilife.be>

Bearbeitungsdauer: 40 Minuten

Appendix

Entwicklung der UML:



Einführung in UML Sequenzdiagramme

Dieses Tutorial entstand mit freundlicher Genehmigung von **YANIC INGHELBRECHT** und seiner Firma Trace Modeler und findet sich im englischen Original unter: <http://www.tracemodeler.com>

Copyright © 2006-2009 Yanic Inghelbrecht. All Rights Reserved.

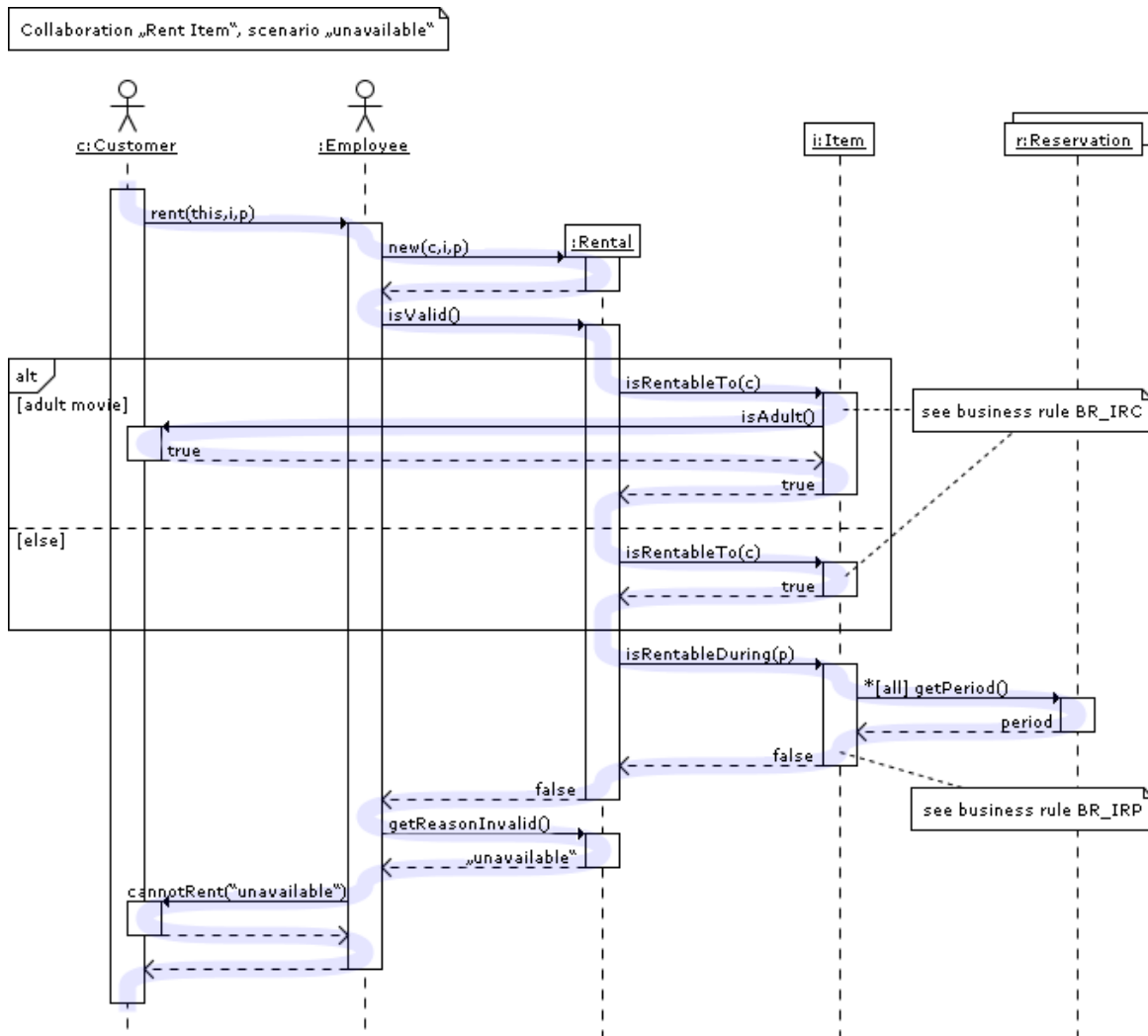
Das Tutorial gibt eine Einleitung in die meist verwendeten Elemente des UML Sequenzdiagramms und erläutert wie man sie benutzt. Alle Diagramme in dieser Anleitung wurden mit der Software „Trace Modeler“ erstellt - ein einfach zu benutzender Editor für UML Sequenzdiagramme.

UML Sequenzdiagramme

UML Sequenzdiagramme werden benutzt, um das Zusammenwirken von Objekten in bestimmten Situationen darzustellen. Eine wichtige Eigenschaft eines Sequenzdiagramms ist der Zeitverlauf der Aktivitäten im Diagramm von oben nach unten: die Interaktion startet im oberen Bereich des Diagramms und endet unten (d. h. tiefer heißt später).

Eine weit verbreitete Anwendung ist die Darstellung der Dynamik in einem Objektorientierten System. Für jede Schlüsselkollaboration werden Diagramme erzeugt die darstellen, wie Objekte in verschiedenen Szenarien dieser collaboration interagieren.

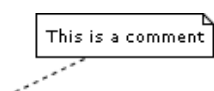
Das Beispiel zeigt ein typisches Sequenzdiagramm basierend auf einem System Use Case - Szenario



Das Diagramm zeigt wie Objekte in der „Miete-Artikel“-Kollaboration interagieren (sich gegenseitig beeinflussen) wenn ein Artikel im angefragten Zeitraum nicht zur Verfügung stehen.

Um zu verdeutlichen wie die Ausführung von einem Objekt zum anderen umschaltet, wurde eine blaue Markierung hinzugefügt, um den Kontrollfluss zu verdeutlichen. Beachten Sie, dass diese Markierung kein Teil des Diagramms darstellt.

Wie in allen UML-Diagrammen werden Kommentare als Rechteck mit einer umgefalteten Ecke dargestellt:



Kommentare werden meist über eine gestrichelte Linie mit Elementen im Diagramm verbunden.

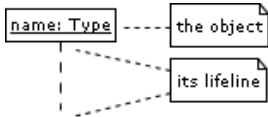
Ziele

Objekte können genau wie Klassen Ziele in einem Sequenzdiagramm sein, das bedeutet, dass Nachrichten zu Ihnen gesendet werden können.

Ein Ziel wird als ein Rechteck mit enthaltenem Text dargestellt. Unter dem Ziel zeichnet man die Lebenslinie, um zu visualisieren, wie lange das Ziel existiert. Die Lebenslinie wird dargestellt als eine vertikale, gestrichelte Linie.

Objekt

Im Folgenden sehen sie die Standarddarstellung für ein Objekt:

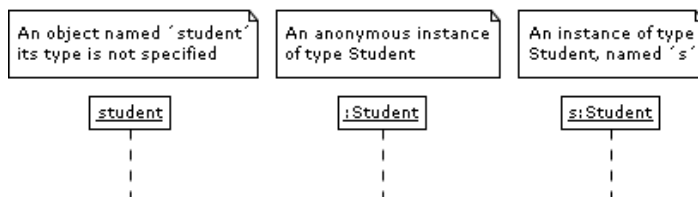


„name“ ist der Name des Objekts im Bezug auf den Inhalt des Diagramms, „Type“ beschreibt den Typ von dem das Objekt eine Instanz darstellt.

Hinweis: Das Objekt muss keine direkte Instanz des angegebenen Typs sein, die Angabe eines Typs von dem das Objekt nur eine indirekte Instanz ist, ist ebenfalls möglich. „Type“ kann also auch ein abstrakter Typ oder ein Interface sein.

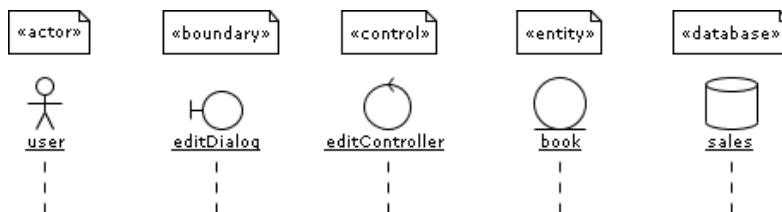
Die Angabe von „name“ und „Type“ ist optional, aber mindestens eine der beiden Angaben sollte gemacht werden.

Beispiele:



Wie in jedem UML-Element, kann man einen Stereotype zu einem Ziel hinzufügen. Einige der häufig benutzten Stereotypen sind:

«actor», «boundary», «control», «entity» und «database». Die Stereotypen können zusätzlich mit entsprechenden Icons dargestellt werden:



Ein Objekt sollte nur benannt werden, wenn mindestens eines der folgenden Kriterien erfüllt ist.

- Sie wollen während einer Interaktion auf das Objekt, als eine Nachrichten-Parameter oder Rückgabewert, referenzieren.
- Sie haben keinen Typ angegeben.
- Es gibt noch andere anonyme Objekte der gleichen Art - ihnen Namen zu geben ist der einzige Weg, die Objekte zu unterscheiden.

Wenn Sie auch den Typ des Objekts festlegen, versuchen Sie lange, aber nicht beschreibende (nichts aussagende) Namen zu vermeiden (verwenden Sie z. B. nicht „aStudent“ für eine Instanz vom Typ Student).

Ein kürzerer Namen trägt die gleiche Menge von Informationen und man vermeidet so ein Übermaß an Informationen innerhalb des Diagramms (z. B. verwenden sie „s“ als Name anstatt „aStudent“).

MultiObjekt

Sie können ein Multiobject verwenden um zu zeigen, wie ein Client mit den Elementen einer Collection interagiert.

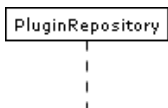
Im Folgenden sehen Sie die Standardnotation des Multiobjects:



Hier können Sie wieder einen Namen und / oder Typ angeben. Beachten Sie jedoch, dass „Type“ die Art der Elemente beschreibt und nicht die Art der Collection an sich.

Klasse

Die Standard Notation für eine Klasse ist:



Nur Klassen - Nachrichten (z. B. in einigen Programmiersprachen, gemeinsame oder statische Methoden) können zu einer Klasse gesendet werden.

Beachten Sie, dass der Text in einer Klasse nicht unterstrichen ist, das unterscheidet die Klasse von einem Objekt.

Nachrichten

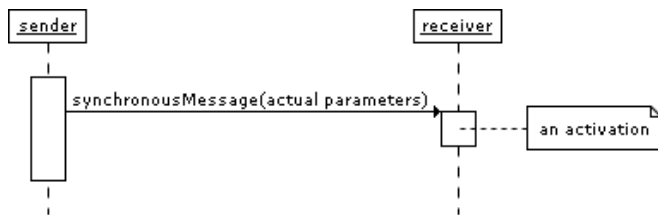
Wenn ein Ziel eine Nachricht zu einem anderen Ziel sendet, wird diese Nachricht als ein Pfeil zwischen den Lebenslinien der beiden Ziele dargestellt.

Der Pfeil beginnt beim Absender und endet beim Empfänger der Nachricht. In der Nähe des Pfeils werden Name und Parameter der Nachricht angegeben.

Synchrone Nachricht

Eine synchrone Nachricht wird verwendet, wenn der Absender wartet, bis der Empfänger die Verarbeitung der Nachricht beendet hat, erst danach arbeitet der Absender weiter (z. B. ein blocking call).

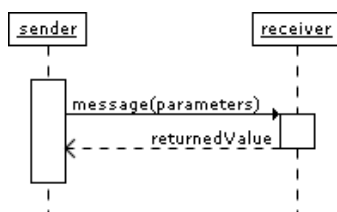
Die meisten Methodenaufrufe in objektorientierten Programmiersprachen sind synchron. Ein geschlossene, gefüllte Pfeilspitze bedeutet, dass die Nachricht synchron gesendet wird.



Die weißen Rechtecke auf einer Lebenslinie heißen Aktivierungen und weisen darauf hin, dass ein Objekt auf eine Nachricht reagiert. Das Rechteck beginnt, wenn die Nachricht empfangen wird und endet, wenn das Objekt die Nachricht abgearbeitet hat.

Wenn eine Nachricht verwendet wird um Methodenaufrufe darzustellen, entspricht jede Aktivierung der Zeitdauer, während der ein Aktivierungseintrag für seinen Anruf auf dem Anruf-Stapel (call-stack) existiert.

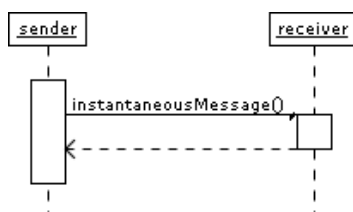
Wenn Sie darstellen wollen, dass der Empfänger die Verarbeitung der Nachricht abgeschlossen hat und die Kontrolle zurück an den Absender gibt, zeichnen Sie einen gestrichelten Pfeil vom Empfänger zum Absender. Optional kann ein Wert, den der Empfänger an den Absender zurück gibt, in der Nähe des Pfeils angegeben werden.



Wenn Sie wollen, dass Ihre Diagramme leicht lesbar sind, zeichnen Sie den Zurück - Pfeil nur, wenn ein Wert zurück gegeben wird, ansonsten lassen Sie den Pfeil weg.

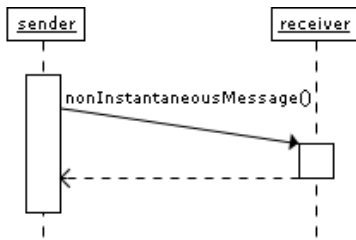
Unmittelbare Nachrichten

Nachrichten werden oft als unmittelbar betrachtet, d. h. die Zeit die benötigt wird um den Empfänger zu erreichen, ist zu vernachlässigen. Zum Beispiel bei einem prozessinternen Methodenaufrufen. Diese Nachrichten werden als waagerechter Pfeil dargestellt.



Manchmal dauert es jedoch eine erhebliche Zeit, bis der Empfänger erreicht wird (natürlich relativ gesehen).

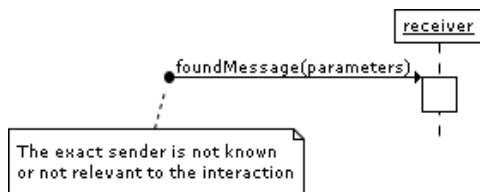
Beispiel: Eine Nachricht innerhalb eines Netzwerks. Eine solche nicht-unmittelbare Nachricht wird als schräger Pfeil dargestellt.



Sie sollten einen schrägen Pfeil nur benutzen, wenn Sie wirklich hervorheben wollen, dass eine Nachricht über einen relativ langsamen Kommunikationskanal versendet wird. (und vielleicht um auf eine mögliche Verzögerung hinzuweisen). Andernfalls benutzen sie den waagerechten Pfeil.

Vorgefundene Nachricht

Eine vorgefundene Nachricht ist eine Nachricht bei der der Aufrufer nicht dargestellt wird. Je nach Kontext bedeuten dies, dass entweder der Absender nicht bekannt ist, oder dass es nicht wichtig ist wer der Absender der Nachricht war. Der Pfeil der eine vorgefundene Nachricht darstellt beginnt mit einem gefüllten Kreis.



Asynchrone Nachrichten

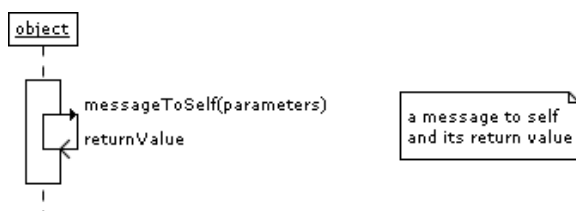
Bei einer asynchronen Nachricht, wartet der Absender nicht bis der Empfänger die Nachricht abgearbeitet hat. Der Absender arbeitet sofort weiter. Nachrichten an einen Empfänger in einem anderen Prozess oder Aufrufe die einen neuen Thread starten, sind Beispiele für asynchrone Nachrichten. Eine offene Pfeilspitze wird verwendet, um das Senden einer asynchronen Nachricht anzuzeigen.



Ein kleiner Hinweis für die Verwendung von asynchronen Nachrichten: Wenn die Nachricht empfangen wurde, arbeiten der Absender und Empfänger gleichzeitig. Die Darstellung von zwei gleichzeitigen Abarbeitungen in einem Diagramm ist allerdings schwierig. In der Regel stellen Autoren nur eine der Abarbeitungen dar, oder zeigen eine nach der anderen.

Nachricht an sich selbst

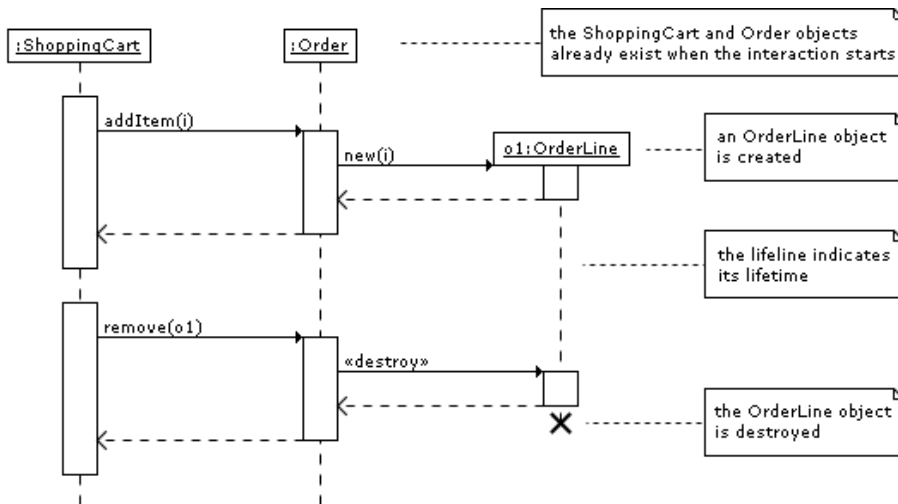
Eine Nachricht, die von einem Objekt an sich selbst gesendet wird kann wie folgt dargestellt werden:



Denken Sie daran, dass es der Zweck eines Ablaufdiagramm ist, die Interaktion zwischen Objekten darzustellen. Überlegen Sie deshalb gut welche „Nachricht an sich selbst“ Sie in ein Diagramm aufnehmen.

Erzeugung und Zerstörung

Ziele, die zu Beginn der Interaktion existieren sind an der Spitze des Diagramms. Alle Ziele, die während der Interaktion erstellt werden, werden weiter unten im Diagramm, zur Zeit Ihrer Erstellung platziert.

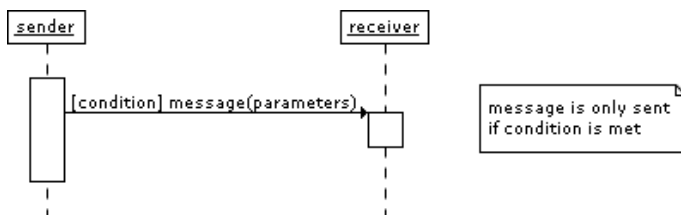


Die Lebenslinie eines Ziels ist so lang wie das Ziel vorhanden ist. Wenn das Ziel während der Interaktion zerstört wird, endet die Lebensader genau zu diesem Zeitpunkt mit einem großen Kreuz.

Bedingte Interaktion

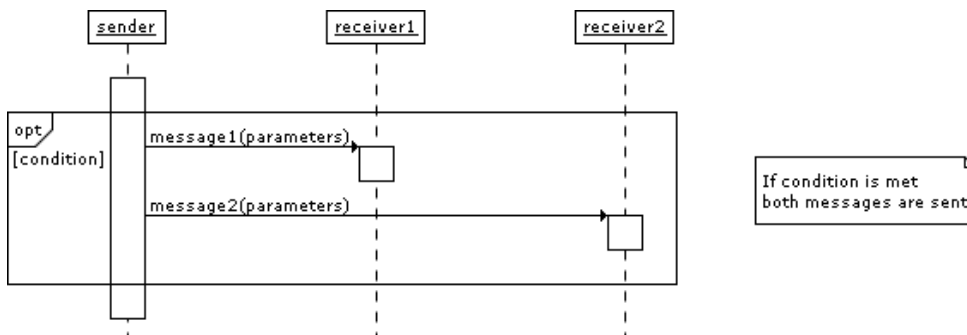
Eine Nachricht kann eine Wächterbedingung enthalten, was bedeutet, dass die Nachricht nur dann gesendet wird, wenn eine bestimmte Bedingung erfüllt ist.

Um eine Wächterbedingung zu definieren, wird die Bedingung in Klammern gesetzt.



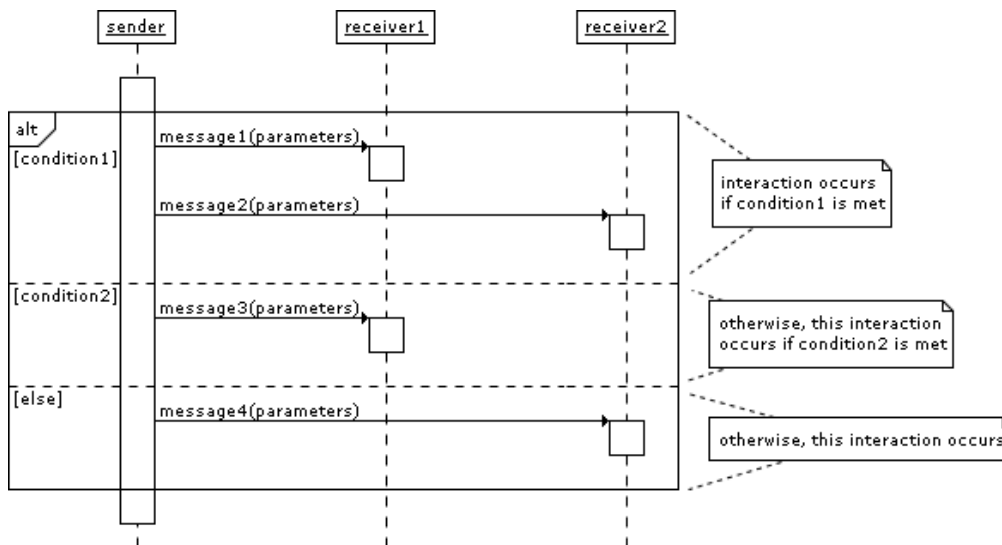
Wenn Sie darstellen wollen, dass mehrere Nachrichten unter der gleichen Wächterbedingung gesendet werden, benutzen Sie ein kombiniertes Fragment mit dem Operator „Opt“.

Das kombinierte Fragment wird als ein großes Rechteck mit einem „Opt“-Operator und einer Wächterbedingung dargestellt und enthält alle bedingten Nachrichten innerhalb dieser Wächterbedingung.



Eine bewachte Nachricht oder „Opt“ kombiniertes Fragment ähnelt dem if-Konstrukt in einer Programmiersprache.

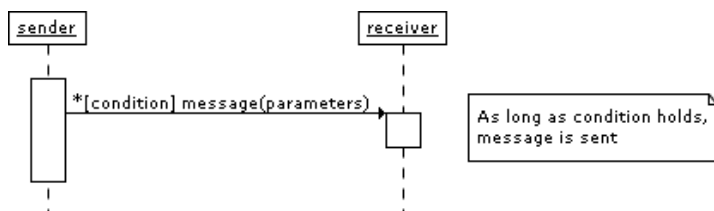
Wenn Sie mehrere alternative Wechselwirkungen zeigen wollen, verwenden Sie ein kombiniertes Fragment mit dem „alt“ Operator. Das kombinierte Fragment enthält einen Operanden für jede Alternative. Jede Alternative hat eine Wächterbedingung und die Interaktion die erfolgt, wenn die Wächterbedingung erfüllt ist.



Da maximal eine Bedingung erfüllt wird, tritt in den meisten Fällen einer der Operanden auf. Ein „alt“ kombiniertes Fragment ähnelt den verschachtelten if-then-else und switch / case-Konstrukten in Programmiersprachen.

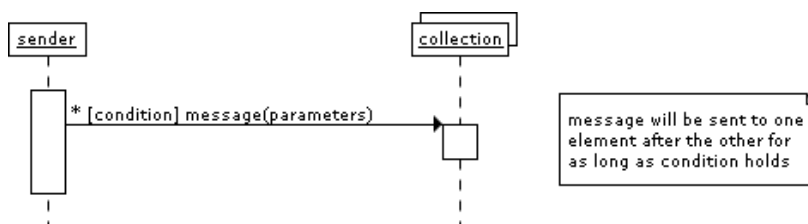
Wiederholte Interaktion

Wenn einer Meldung ein Stern (das „*“-Symbol) vorangestellt ist bedeutet dies, dass die Nachricht wiederholt gesendet wird. Eine Wächterbedingung bestimmt, ob die Nachricht (wieder) gesendet werden soll oder nicht. Solange die Bedingung gilt, wird die Nachricht wiederholt.



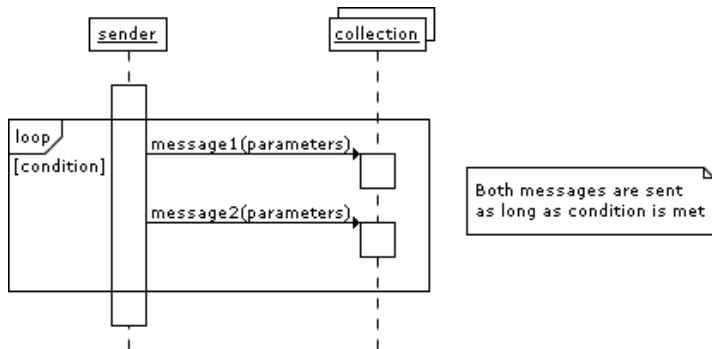
Die oben dargestellte Interaktion von wiederholtem Senden derselben Nachricht an das selbe Objekt, ist nicht sehr nützlich, es sei denn Sie wollen eine Art „Polling-Szenario“ darstellen.

Eine gebräuchlichere Nutzung der Wiederholung ist das Senden der gleichen Nachricht an verschiedene Elemente in einer Sammlung. In einem solchen Szenario, ist der Empfänger der wiederholten Nachricht ein Multiobjekt und die Wächterbedingung ist die Bedingung, welche die Wiederholung kontrolliert.



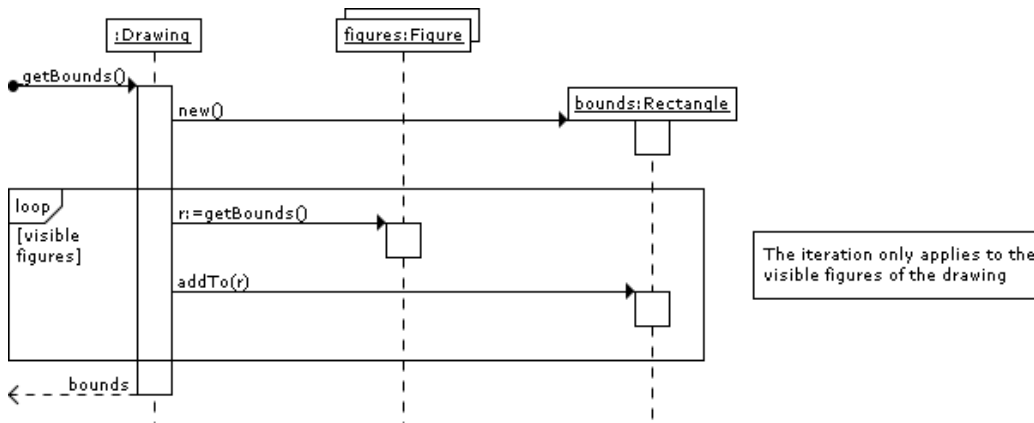
Dies entspricht einer Iteration über die Elemente in der Sammlung, bei denen jedes Element die Nachricht erhält. Für jedes Element, wird die Bedingung ausgewertet, bevor die Nachricht gesendet wird. In der Regel wird die Bedingung aber als Filter benutzt, der Elemente aus der Sammlung auswählt (z. B. „Alle“, „Erwachsene“, „neue Kunden“ als ein Filter für eine Sammlung von Person-Objekte). Nur die durch den Filter ausgewählten Elemente werden die Nachricht erhalten.

Wenn Sie darstellen wollen, dass mehrere Nachrichten in der gleichen Iteration gesendet werden, kann ein kombiniertes Fragment mit einem „Schleifen“-Operator verwendet werden. Der Operator des kombinierten Fragments ist „Loop“ und die Wächterbedingung ist die Voraussetzung, um die Wiederholung zu kontrollieren.



Wenn der Empfänger einer wiederholten Nachricht wieder eine Sammlung (Collection) ist, wird die Bedingung allgemein dazu benutzt, einen Filter für die Elemente festzulegen.

Will man z. B. zeigen, dass die Umrandung einer Zeichnung auf den Umrandungen seiner sichtbaren Figuren beruht, könnte man das folgende Sequenzdiagramm zeichnen:



Mehrere Dinge in diesem Beispiel sind einer Anmerkung Wert:

- Eine lokale Variable „r“ wurde eingeführt, um das Ergebnis des Aufrufs von `getBounds()` aufzunehmen.
- Durch die Benennung des Ergebnis-Rechtecks „bounds“ ist die Einführung einer zusätzlichen lokalen Variable nicht notwendig.
- Die Schleifen-Bedingung wird als ein Filter auf den Elementen der Collection verwendet.

UML - Werkzeug - Unterstützung

Bei allen UML Diagramm-Typen, ist der Sequenz-Diagramm-Typ derjenige, bei dem es am Wichtigsten ist, das richtige UML - Werkzeug zu wählen. Der Grund dafür ist, dass Sie sehr wenig Freiheit haben, wenn es darum geht Elemente in einem Sequenz-Diagramm zu positionieren:

- Einige Elemente müssen in ein bestimmtes Bereich abgelegt werden,
- Einige Elemente müssen andere einschließen.
- Viele Elemente sind miteinander verbunden.
- Die meisten Elemente haben eine feste Ausrichtung.
- Die gitterähnliche Struktur fordert praktisch einen gleichmäßigen Abstand.
- Es gibt oft Situationen bei denen sich Elemente auf eine unschöne Weise überlappen.

Sie benötigen ein Werkzeug, das für das Erstellen von Sequenz-Diagrammen entworfen wurde. Benutzen sie kein gewöhnliches Zeichenprogramm, Sie werden sonst Stunden mit dem Verbinden, Größenanpassungen und dem Entwerfen von Formen verschwenden.

Würden Sie davon ausgehen, dass aktuelle UML-Werkzeuge sie bei diesen Ummengen an Einschränkungen beim Entwurf unterstützen? Denken Sie nochmal darüber nach.

Die meisten UML-basierten Werkzeuge bieten nur grundlegende Unterstützung für Sequenz-Diagramme an und haben einen geringen Nutzen. Obwohl sie eine Verbesserung gegenüber generellen Editoren darstellen, bieten sie wenig Unterstützung in den Punkten Layout und Sie werden immer noch viel Zeit beim hin und her bewegen von Elementen verlieren.

Wenn Sie ein Werkzeug bewerten wollen, finden Sie heraus wie es sich verhält, wenn Sie ein vorhandenes Diagramm ändern. Fügen Sie Dinge hinzu, bewegen Sie Elemente und schauen Sie sich das resultierende Diagramm an. Ist das noch ein gut aussehendes Diagramm oder müssen Sie das das Layout nochmals handisch anpassen?



Eine Checkliste von Dingen, die Sie ausprobieren sollten, ...

... wenn Sie ein Werkzeug für Sequenzdiagramme testen.




- Fügen Sie eine neue Nachricht ein. -> Mussten Sie Pfeile und Aktivierungen mit der Hand verbinden?
- Bewegen Sie ein Ziel nach links oder rechts. -> Sind die Nachrichtenpfeile immer noch mit der richtigen Seite der Aktivierungen verbunden?
- Fügen Sie in der Mitte des Diagramms eine neue Nachricht ein. -> Wurden die vorhandenen, darunterliegenden Elemente automatisch bewegt, um Platz für die neue Nachricht zu machen?
- Ändern Sie den Empfänger einer Nachricht. -> Haben sich die Aktivierungen und Pfeile entsprechend angepasst? Selbst wenn Sie es zu einer Nachricht auf sich selbst gemacht haben?
- Bewegen Sie eine Nachricht oder eine Aktivierung nach oben oder unten. -> Wurden sie und die Elemente um sie herum entsprechend angepasst oder mussten Sie das selbst tun?
- Bewegen Sie eine Nachricht in und aus ein kombiniertem Fragment. -> Wurde das Fragment automatisch in der Größe angepasst und bewegte es sich Zusammenhang mit seinem Inhalt?
- Scrollen Sie nach unten, so dass die Ziele nicht mehr sichtbar sind. -> Zeigt das Werkzeug Hinweise darauf an, welche Lebenslinie zu welchem Ziel gehört?
- Ändern Sie den Namen einer Nachricht. -> Konnten Sie den Namen direkt im Diagramm

selbst ändern?

Wählen Sie das Werkzeug mit den besten, automatischen Layout-Eigenschaften aus, es wird Ihnen eine erhebliche Menge Zeit einsparen.


YANIC INGHELBRECHT, der Entwickler des Trace Modeler beschreibt seine Erfahrungen auf der Suche nach dem passenden Werkzeug wie folgt: „*I never quite found what I was looking for and ended up creating my own. Trace Modeler supports all of the above and has some  really neat layout features and benefits that I haven't seen anywhere else, I invite you to watch the  demo and then try it yourself!*“


Hinweise des Autors auf weiterführende Links

Diese schnelle Einführung stellte nur einige wenige der möglichen Konstrukte vor, auf die Sie in einem UML Sequenz-Diagramm stoßen können. Ein weiterer Artikel auf  www.tracemodeler.com stellt vor wann und wie man Sequenz-Diagramme in Projekten benutzt. Schauen Sie dafür in den  Artikel-Abschnitt oder abonnieren sie den  [News-Feed](#), um Benachrichtigungen über neue Artikel und Inhalte zu erhalten.

Wenn Sie anfangen alles herauszufinden, was es über UML-Sequenz-Diagramme zu wissen gibt, behalten Sie im Hinterkopf, dass es wichtiger ist, zu wissen wann und wie man Sie benutzt, als alle möglichen Konstrukte der UML Spezifikation zu kennen.

Für praktische Tipps, wie Sie Ihre eigenen Diagramme verbessern, werfen Sie einen Blick auf die Artikelserie

 [Verbessern von Diagrammen](#) Jede Episode bespricht und verbessert ein wirkliches im Web gefundenes Sequenz-Diagramm-Beispiel.

Werfen Sie einen Blick in die  [Galerie](#), sie enthält viele Beispiel-Sequenz-Diagramme.

Copyright © 2006-2009 Yanic Inghelbrecht. All Rights Reserved.