# Proseminar - Model-Driven Development - SS05

# Modellierung mit UML 2

## Beitrag der UML 2 zur modellbasierten Entwicklung
## Vorstellung der UML 2 Diagrammtypen

Marc Giombetti

TU Kaiserslautern

**Zusammenfassung** Im Rahmen dieser Arbeit wird die UML als Modellierungssprache in den Kontext der modellbasierten Entwicklung eingeordnet. Die zentralen Begriffe Modell, Diagramm, Sicht und die Zusammenhänge, die zwischen diesen Begriffen bestehen, werden in Abschnitt 1.3 vorgestellt und erklärt. Anhand des Beispiels eines Use-Case-Diagramms (Abschnitt 2.1) und eines Klassendiagramms (Abschnitt 2.2) wird sowohl ein dynamisches, als auch ein statisches Diagramm vorgestellt. Da ein Schwerpunkt der modellbasierten Entwicklung, die Transformation von Diagrammen ist, werden hier zwei Ansätze verfolgt und Beispiele gegeben. Zum einen wird in Abschnitt 2.3 die Transformation eines Use-Case-Diagramms in ein Klassendiagramm, diagrammübergreifend anhand eines vereinfachten Metamodells erklärt.
Zum andern wird in Abschnitt 2.4 die Transformation von einem Modell in Programmiercode, hier am Beispiel der Transformation eines Klassendiagramms in Java-Code, gezeigt. Obwohl die Implementierung eines Modells immer an eine gewisse Plattform angepasst werden muss, ist es klar, dass Konzepte langlebiger als Realisierungen sind. In dem Kontext wird in Abschnitt 2.5 das Konzept der PIM und PSMs vorgestellt, welches Technologiewandel beherrschbar macht.
Abschliessend wird dann in Abschnitt 2.6 auf einige Probleme bei der modellbasierten Entwicklung und der automatisierten Quellcodeerzeugung eingegangen.

# 1 Beitrag der UML 2 zur modellbasierten Entwicklung

## 1.1 Was ist UML ?

Die UML (Unified Modeling Language) ist heutzutage der de facto Standard zur Modellierung, Dokumentierung, Spezifizierung und vor allem Visualisierung von immer komplexer werdenen Softwaresystemen und -produkten.[1]
Die erste Version der UML entstand durch die Zusammenführung praxisrelevanter Methoden und Schemen verschiedener Autoren wie Boch, Rumbough und Jacobson, deren Modellierungstechniken damals einen hohen Marktanteil und eine hohe Verteiltheit aufzeigten.

Mit der Entwicklung der Version 1.x hatten die Entwickler ein mächtiges Werkzeug erschafft, welches sehr viele Konstrukte besaß. Für einige Anwendungsbereiche, wie z.B. Serverprogrammierung und Echtzeitanwendungen, hingegen bot sie jedoch nur sehr eingeschränkte Modellierungsmöglichkeiten. Mit der Version 2 wurden dann einige Diagramme erweitert, umstruktiert und neu in den Sprachraum aufgenommen, um somit mehr Präzision zu ermöglichen und Übersichtlichkeit zu schaffen. Die Präzisionssteigerung bei der Modellierung wird vor allem durch den erweiterten Gebrauch der Object Constraint Language (OCL) ermöglicht. Die OCL ist eine einfache formale Sprache, mit der UML-Modellen weitere Semantik hinzugefügt werden kann, die mit den übrigen UML-Elementen nicht oder nur sehr umständlich ausgedrückt werden könnte [4].

Das UML Metamodell wurde komplett überarbeitet, um somit die Kombination der verschiedenen Diagramme, d.h. die Verknüpfungsmöglichkeiten unter ihnen zu erleichtern. Neben der Möglichkeit, eine grobe skizzenartige Modellierung zu verwenden, bei der von lediglichen technischen Details abstrahiert werden kann, ist die UML ein Werkzeug, das eine sehr präzise, und dem jeweiligen Anwendungsbereich angepasste Modellierung ermöglicht.

Im folgenden werden wir die UML in das Konzept der modellbasierten Entwicklung einordnen und in Abschnitt 1.3 wichtige Begriffe der modellbasierten Entwicklung erklären.

## 1.2 Einordnung der UML in die modellbasierte Entwicklung

Modelle sind ein natürlicher Weg um komplexe Vorgänge und Systeme zu verstehen, zu untersuchen und zu planen, um sie dann in die Realität zu transformieren [5].

Zu einem erfolgreichen Projekt gehört also ein guter Plan und so entstand eine Menge verschiedener Modellierungsmethoden zum Entwerfen von Softwaresystemen. Doch diese Vielzahl unterschiedlicher Modelle und Modellnotationen, die alle für einen konkreten Einsatzzweck geschaffen wurden, bereiten in der Praxis bei Ihrer Zusammenführung oft Inkonsistenzen, da jede Notation „anders“ ist und jeweils spezifische Informationen enthält. Es kam also seitens der Object Management Group (OMG) die Idee auf, die wichtigsten Modellnotationen

in einer Modellierungssprache so zusammenzufassen, dass eine explizite Kombinierung der Modelle möglich werden sollte. Die OMG ist ein 1989 gegründetes Konsortium, das sich mit der Entwicklung von Standards für die herstellerunabhängige systemübergreifende Objektorientierte Programmierung beschäftigt. Die OMG hat über 800 Mitglieder, darunter Firmen wie IBM, Apple und Sun [2]. Neben der UML, hat die OMG auch das Model-Driven Architecture Konzept (MDA) und die Common Object Request Broker Architecture (CORBA) erschaffen, die das Erstellen von Verteilten Anwendungen in heterogenen Umgebungen ermöglicht.

Wichtiges Stichwort in Bezug auf UML ist der Begriff *unified*. Im Gegensatz zu ihren Vorgängerversionen, die sich vor allem mit der Modellierung von typischen Softwaresystemen beschäftigt haben, wurde die UML 2 explizit für ein breiteres Anwendungsfeld entworfen. Sie ermöglicht somit sowohl die Modellierung von Geschäftsprozessen, Testfällen, Prozeduren, Organisationen und Datenstrukturen, als auch ansatzweise die Modellierung von Abläufen in technischen Systemen und Echtzeitsystemen. Durch die Vereinigung von oftverwendeten und gut konzipierten und kombinierbaren Diagrammen, ermöglicht sie es, fast jedes softwaretechnische nur denkbare Modell zu realisieren.

Die UML soll durch die Regruppierung und Vereinigung eines Teils dieser unterschiedlichen Modellierungstechniken eine Basis für effektive und konsistente Modelle schaffen. Die UML ist in vielen Planungsphasen des Softwareentwicklungsprozess einsetzbar: von der ersten groben Planung bis zur detaillierten Endausarbeitung, d.h. sowohl in der Analyse- als auch in der Entwurfsphase. Durch die aufeinander aufbauende Sprachdefinition braucht man zum Entwickeln eines Modells für einen gewissen Zweck nur Kenntnis über den Sprachkern und die jeweiligen für den Zweck geeigneten Sprachpakete. Dies ermöglicht dem Entwickler bzw. Softwareanalytiker einen schnellen Einstieg in UML, da man sich nur die für sein Projekt oder seinen Anwendungszweck relevanten Aspekte der UML aneignen muss.

### 1.3 Modell - Diagramm - Sicht

Wichtig in Hinsicht auf die modellbasierte Entwicklung ist auch die Unterscheidung zwischen dem Modell, den Diagrammen und den sogenannten Sichten zu machen.

Ein Modell ist ein vereinfachtes Abbild der Wirklichkeit und ermöglicht es wesentliche Eigenschaften hervorzuheben sowie unwichtige Aspekte außer Acht zu lassen. Ein Modell bildet die Zielvorstellung und die Grenzen dessen, was automatisch berechnet werden kann, und ermöglicht die Voraussage künftigen Verhaltens.

Ein Diagramm ist eine grafische Darstellung von Daten oder Informationen. In unserem Fall handelt es sich um eine visualisierte Teilmenge des Modells.

Eine Sicht ist eine gewisse Perspektive auf ein Modell. Sie stellt nur die für den jeweiligen Standpunkt wichtigen Aspekte dar und abstrahiert von den irrelevanten. Die Sicht „verknüpft“ somit das Modell mit dem Diagramm.

Ein Modell muss aber nicht zwingenderweise mittels Diagrammen dargestellt werden. Im Rahmen der UML wird zum Abspeichern der Modellinformationen in textueller Form ein XML-Derivat namens „XML Meta Interchange“(XMI) verwendet. Dieser offene und anbieterneutrale Standard der OMG ermöglicht den Datenaustausch von Objekten auf der Basis von Meta-Metamodellen nach der Meta-Object Facility (MOF).

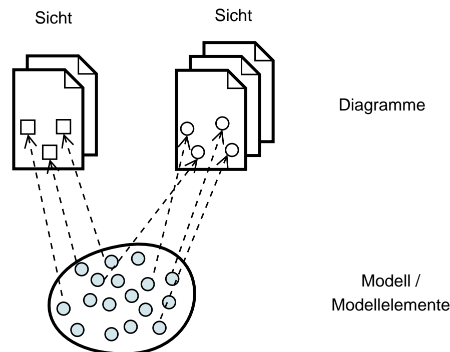


**Abbildung 1.** Modell - Diagramm - Sicht [5]

Durch ein gewisses Modellierungskonzept, wie z.B. der UML, ist es möglich das „abstrakte“ Modell, bzw. einige Modellelemente in Form von Diagrammen darzustellen. Die Diagramme müssen auch nicht immer alle Informationen enthalten die bei der Umsetzung nötig sind. Dies stellt sicher, dass ein Diagramm nicht zu überladen wird und somit sein Ziel verfehlt. Dass entspricht dem vorhin angedeuteten Prinzip der Abstraktion, welches eine sehr hohe Bedeutung in der Informatik hat. Ein weiterer wichtiger Punkt ist die Konsistenz zwischen den Diagrammen und dem Modell, d.h. dass die im Diagramm dargestellten Informationen auch tatsächlich im Modell vorkommen.

Abbildung 1 zeigt, dass verschiedene Modellelemente als Diagramme oder als Gruppe von Diagrammen dargestellt werden können. Die Diagramme ermöglichen eine präzise Sicht auf einen Teil des Systems und stellen somit verschiedene „Abstraktionsebenen“ und Systemtiefen dar. Modelle die verschiedene Sichten darstellen, sind keine Verfeinerungen voneinander.
Wie vorhin schon angedeutet, wird eine Notation verwendet die eine Erweiterung oder Einschränkung der Sprache an den jeweiligen Einsatzbereich ermöglicht. Bei dieser Notation handelt es sich um die „physikalische“ Darstellung des Modells in Form von Diagrammen. Das UML-Diagramm wird somit zur visuellen Darstellung des Modells.

Bis jetzt haben wir uns ausschließlich mit dem Konzept der UML und der Einordnung in die modellbasierte Entwicklung beschäftigt. Im folgenden soll nun der UML Sprachraums mit seinen Diagrammen vorgestellt werden.

## 2 Vorstellung der UML 2 Diagrammtypen und der Möglichkeit zur Codegenerierung

Die UML 2.0 definiert 13 verschiedene Diagrammtypen, die sich in die 2 Hauptkategorien (statische) Strukturdiagramme und (dynamische) Verhaltensdiagramme unterteilen lassen. Verschiedene Verhaltensdiagramme kann man zusätzlich noch in der Obermenge Interaktionsdiagramme zusammenfassen. Abbildung 2 zeigt die Gliederung der Diagrammtypen im UML 2 Sprachraum.

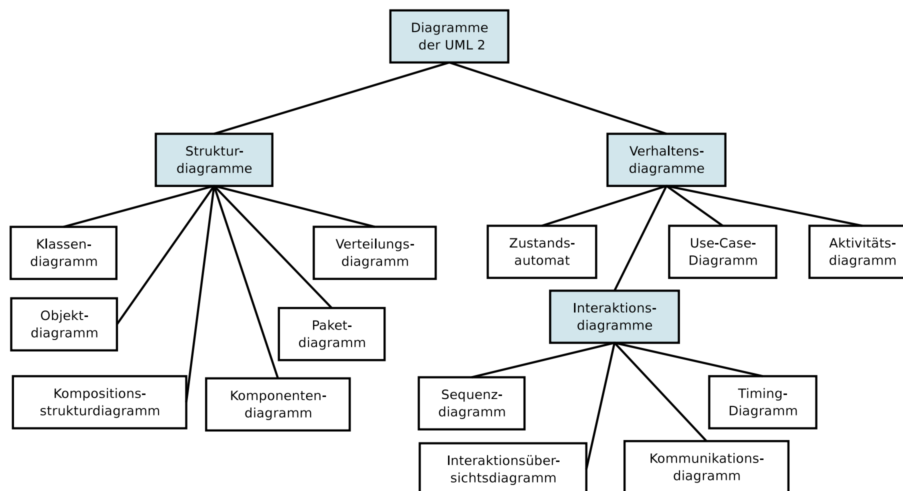


**Abbildung 2.** Diagramme der UML 2

Neu in der UML 2 sind das Kompositionsstrukturdiagramm und das Timing-Diagramm.

Beim Kompositionsstrukturdiagramm handelt es sich um ein Diagramm das die Strukturierung der einzelnen Architekturkomponenten eines Systems ermöglicht und ihr Zusammenspiel veranschaulicht. Es stellt somit die interne Struktur eines Classifiers dar. Gleichzeitig gestattet es die Dokumentation von Designentscheidungen, sowie den Einsatz von Entwurfsmustern.

Das Timing-Diagramm, als zweiter Neuling, wird schon seit längerer Zeit in der Elektrotechnik verwendet um das zeitliche Verhalten digitaler Schaltungen zu beschreiben. In der UML 2 ermöglicht es die präzise Analyse und Darstellung des zeitlichen Verhaltens von Klassen, Akteuren, Komponenten, Schnittstellen usw.
Die Vorstellung aller Diagrammtypen würde den Rahmen dieser Arbeit sprengen, deshalb wollen wir uns in Abschnitt 2.1 näher mit dem Use-Case-Diagramm

als Analysediagramm und in Abschnitt 2.2 mit dem Klassendiagramm als Designdiagramm beschäftigen[1].

## 2.1 Das Use-Case-Diagramm

Das Use-Case-Diagramm (dt: Anwendungsfalldiagramm) soll darüber Auskunft geben was das zu entwickelnde System aus Sicht der Nutzer überhaupt leisten soll. Ab der UML 2 wird das Use-Case-Diagramm den (dynamischen) Verhaltensdiagrammen zugeordnet. Es handelt sich um ein einfaches Diagramm, das vor allem in der Analyse eingesetzt wird, und somit auch intuitiv vom Kunden verstanden werden kann. Die Nutzer des Systems werden im UML Jargon Akteure genannt und als Strichmännchen gezeichnet. Ein Akteur kann eine Person, ein Fremdsystem oder ein zeitliches Ereignis sein. Ab der UML 2 sollen zeitliche Ereignisse als Akteure aber vermieden werden. Akteure müssen sich immer außerhalb des Systems befinden.

Beim Entwurf eines Softwaresystems ist es bekannterweise sehr wichtig genau zu wissen für wen das System was leisten soll und wo die Systemgrenzen liegen. Eine genaue Identifizierung der Akteure ist somit genau so wichtig, wie die Auflistung aller fachlichen Anforderungen an das System. Genau hier kommen Use-Case-Diagramme ins Spiel, da sie das System, die Akteure und die jeweiligen Assoziationen zwischen den Akteuren und dem System darstellen können. Es muss sich immer um binäre Assoziationen handeln, mehrwertige Assoziationen sind nicht erlaubt. Oftmals wird man sich durch die Darstellung mittels Use-Cases noch genauer über die an das System zu stellenden Forderungen, die betroffenen Akteure und die Interaktion der Akteure mit dem System bewusst. Ein Use-Case ist an sich eine Sammlung von Aktionen die in einer gewissen Reihenfolge abzulaufen haben. Use-Cases werden gewöhnlich als Ellipse gezeichnet und müssen einen Namen tragen.
Use-Case-Diagramme sollen hier nicht formell, sondern anhand des in Abbildung 3 dargestellten Use-Case-Diagramms einer Einweihungsfeier erklärt werden, ohne auf alle vorgesehenen Notationselemente für Use-Cases eingehen zu wollen.

Bei diesem Beispiel soll gezeigt werden, dass ein Use-Case-Diagramm die reale Welt und nicht gezwungenerweise ein System aus Hard- und Software modelliert.

**Beispiel:** Der Rahmen stellt bei Use-Case-Diagrammen die Systemgrenzen dar. Innerhalb des Rahmens stehen der Name des Systems und die jeweiligen Use-Cases. Die Akteure befinden sich wie schon erwähnt immer außerhalb des Systems und somit außerhalb des Rahmens. Alle Akteure sind mit einer Linie mit dem jeweiligen Use-Case verbunden. Aus dem Diagramm geht hervor, dass der Akteur *Gast* sowohl *Tanzen*, (sich) *Unterhalten* und auch *Essen* und *Getränke Nachschub anfordern* kann. Der *Gastgeber* der an sich auch ein *Gast* ist, erbt somit diese Eigenschaften (Vererbungspfeil) vom *Gast* und kann zusätzlich noch

[1] Beachten Sie: Use-Cases können auch als Designdiagramme und Klassendiagramme als Analysediagramme verwendet werden.

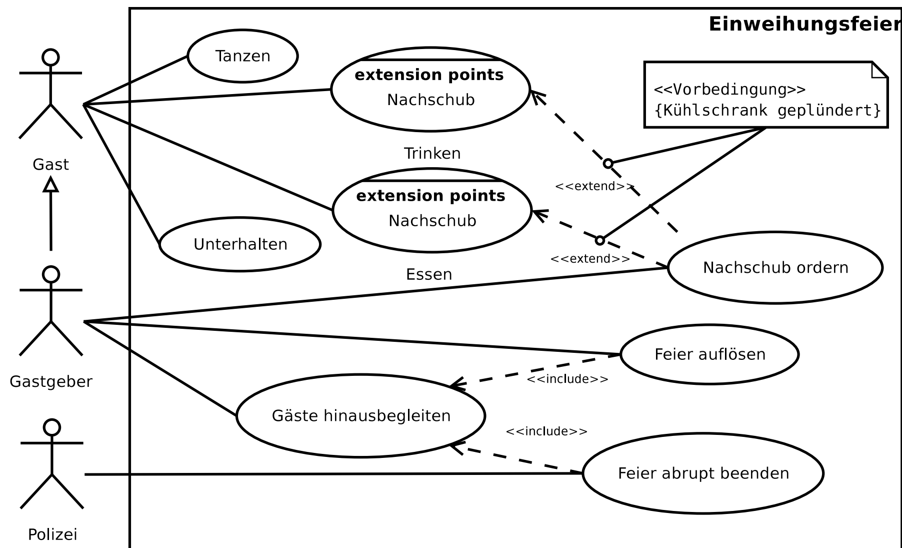


**Abbildung 3.** Use-Cases auf einer Einweihungsfeier [3]

*Nachschub ordern* und die *Gäste hinausbegleiten*. Die *Polizei*, als dritter Akteur kann die *Feier abrupt beenden*.

In diesem Diagramm sind die Use-Cases *Nachschub Trinken* und *Nachschub Essen extension points* und durch eine gestrichelte gerichtete Kante mit *Nachschub ordern* verbunden. Die Pfeilspitze zeigt in Richtung *extension point* und die Kante ist mit ≪*extend*≫ markiert. An die ≪extend≫-Beziehungen kann immer eine Bedingung geknüpft werden, die dann in einem Notizzettel angegeben wird. In diesem Fall bedeutet das, dass falls ein *Gast Nachschub anfordert* und der *Kühlschrank geplündert* wurde, dass der *Gastgeber* dann *Nachschub ordern* kann, aber nicht muss. Für die ≪include≫-Beziehung ist die Notation die gleiche wie für die ≪extend≫-Beziehung, nur dass die Kante mit ≪*include*≫ markiert ist. Der wesentliche Unterschied zur ≪extend≫-Beziehung liegt allerdings darin, dass die ≪include≫-Beziehung jedes Mal aufgerufen werden muss. In unserem Beispiel bedeutet das, dass wenn die *Feier aufgelöst* wird, oder *abrupt aufgelöst* wird, der *Gastgeber* die *Gäste* immer hinausbegleitet. Da es sich um eine Muss-Beziehung handelt, macht die Angabe einer Bedingung bei der ≪include≫-Beziehung keinen Sinn[2].

Ähnlich wie für das Use-Case-Diagramm, soll das Klassendiagramm nun im weiteren Verlauf näher vorgestellt werden.

---

[2] Zyklische Includes im Sinne von A ≪include≫ B und B ≪include≫ A sind nicht möglich.

## 2.2 Das Klassendiagramm

Das Klassendiagramm soll darüber Auskunft geben, wie die Daten und das Verhalten des zu entwerfenden Systems strukturiert sind. Es ist im Gegensatz zum Use-Case-Diagramm ein statisches Diagramm und zeigt die statischen Beziehungen zwischen Klassen. Klassendiagramme sind der älteste Diagrammtyp der UML und stellen den Kern der Modellierungssprache dar.

Klassendiagramme werden in jeder Projektphase verwendet. Sie können von der ersten Skizzierung des Systems bis zur Modellierung der kleinsten Details verwendet werden. Man unterscheidet hierbei zwischen konzeptuell-analytischer Modellierung und logisch design-orientierter Modellierung. Die konzeptuell-analytische Modellierung wird in einer frühen Phase verwendet, bei der es darum geht Zusammenhänge im System übersichtlich zu strukturieren. Die logisch design--orientierte Modellierung hingegen ist viel präziser und ist näher an die technischen Umsetzung gebunden.

**Beispiel:** Genau wie beim Use-Case-Diagramm, soll hier das Klassendiagramm anhand eines Beispiels aus der realen Welt (Abbildung 4) erklärt werden. In die-

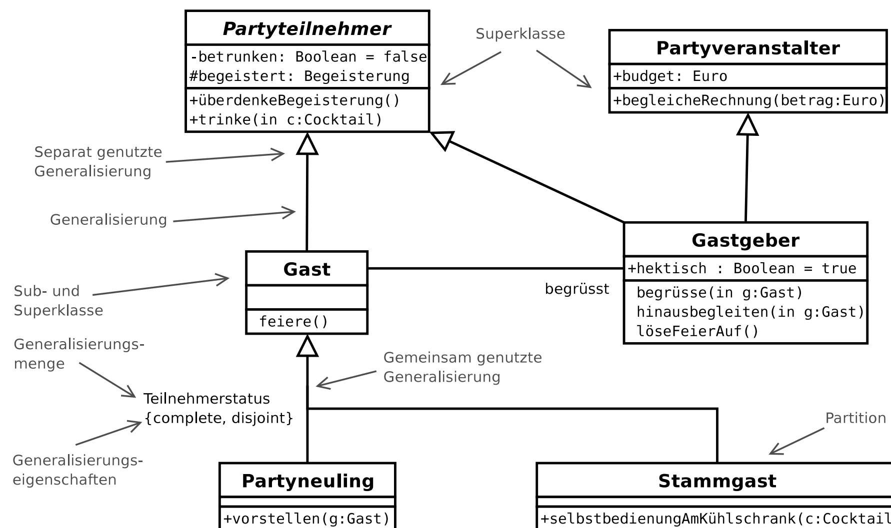


**Abbildung 4.** Klassendiagramm einer Party [3]

sem Diagramm existieren 6 Klassen, eine abstrakte Klasse *Partyteilnehmer* von

der keine Objekte erzeugt werden können und fünf „normale“ Klassen nämlich *Partyveranstalter*, *Gast*, *Gastgeber*, *Partyneuling* und *Stammgast*. Die jeweiligen Klassen besitzen gewisse Attribute und Methoden. Die Klassen *Partyteilnehmer* und *Partyveranstalter* fungieren hier als Superklasse, die keine Klassen spezialisieren, sondern nur generalisieren, d.h. nicht spezialisiert werden. Diese Generalisierungsbeziehung, besser unter dem Namen Vererbung bekannt, besteht z.B. zwischen *Gast* und *Partyteilnehmer*. Der *Gast* besitzt die Methode *feiern()* aber keine Attribute, erbt jedoch alle Attribute und Methoden vom *Partyteilnehmer*. Der *Gastgeber* erbt, z.B. sowohl von der Klasse *Partyteilnehmer*, als auch von der Klasse *Veranstalter*. Dies bezeichnet man als Mehrfachvererbung und bereitet bei der Umsetzung oft Schwierigkeiten. Auf diese Schwierigkeiten werden wir später noch näher eingehen.

Die Klasse *Gast* ist hier Sub- und Superklasse, d.h. sie erbt und vererbt Methoden und Attribute. Zwischen den Klassen *Gast* und *Gastgeber* besteht die Assoziation *begrüßt*. Sie ist ungerichtet und wird durch einen einfachen Strich zwischen den Klassen notiert. (Der *Gast* kann den *Gastgeber* begrüßen und umgekehrt). Die Klassen *Partyneuling* und *Stammgast* besitzen eine gemeinsame Generalisierung (Notation) nämlich die Klasse *Gast* und durch eine weitere Vererbung implizit die Superklasse *Partyteilnehmer*.

Bei einer Generalisierungsbeziehung kann der Abstraktionsschritt angegeben werden, welcher der Generalisierung zugrunde liegt. Zusätzlich können Generalisierungseigenschaften angegeben werden. Hier wird *complete* und *disjoint* verwendet.
*Complete* bedeutet, dass die Vereinigung aller Partitionen alle im Modellkontext sinnvollen und denkbaren Spezialisierungen eines Typs vereinbart.
*Disjoint* bedeutet, dass keine Instanz eines Subtypen gleichzeitig Instanz eines anderen Subtypen ist. Sprich, hier ist es ist nicht möglich, dass jemand *Partyneuling* und *Stammgast* zugleich ist.

Nach der Vorstellung vom Use-Case-Diagramm als Analysemodell und dem Klassendiagramm als Designmodell stellt sich die Frage ob eine Möglichkeit besteht, beide Modelle ineinander zu transformieren. Auf diese Frage wollen wir in Abschnitt 2.3 näher eingehen.

## 2.3 Metamodell zur Transformation eines Analysemodells in ein Designmodell

Die Verbindungen zwischen dem Use-Case-Diagramm (siehe Abschnitt 2.1) und dem Klassendiagramm (siehe: Abschnitt 2.2) sollen hier anhand eines vereinfachten Metamodells dargestellt werden. [3]

[3] Dieses Metamodell stellt keine Ansprüche auf Vollständigkeit, sondern soll dem Leser vermitteln, dass sich auf einer höheren Ebene (Metaebene) Zusammenhänge darstellen lassen, welche die Voraussetzung zur Transformation von Modellen sind.

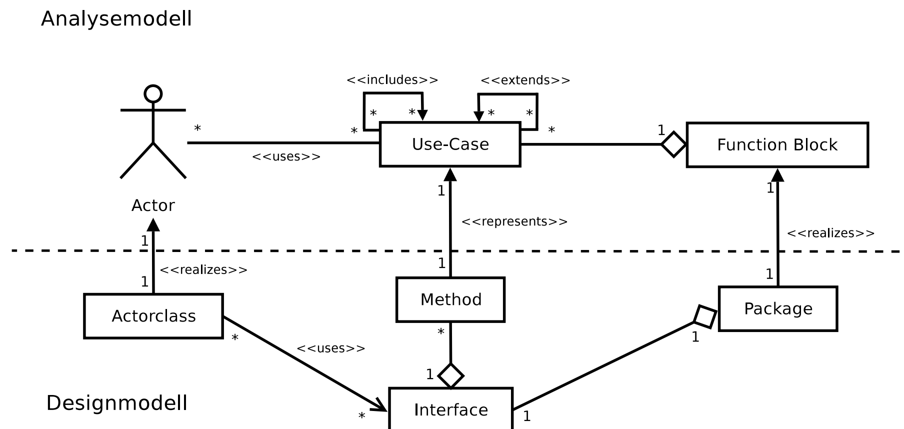


**Abbildung 5.** Stark vereinfachtes Metamodell zur Transformation eines Analyse- in ein Designmodell

**Beispiel:** Das in Abbildung 5 dargestellte Metamodell ist horizontal in einen Analyse- und einen Designteil aufgespaltet. Aus dem Analyseteil geht hervor, dass ein Akteur (*Actor*) *Use-Cases* benutzt (≪*uses*≫) und ein *Function Block* ein oder mehrere *Use-Cases* besitzt. Function Blocks stellen, wie wir beim Use-Case-Diagramm gesehen haben, ein (Teil-)System und dessen Grenzen dar. Eine Use-Case kann dann weitere Use-Cases einbinden und erweitern (≪*includes*≫,≪*extends*≫)[4]. Das Designmodell besteht hier aus einem *Package* das ein oder mehrere *Interfaces* hat. Ein Interface wird von einem oder mehreren Akteurklassen (*Actorclass(es)*) benutzt (≪uses≫). Ein Interface besteht hier im Metamodell Sinne aus einer oder mehreren Methoden und soll nicht mit der Java-Implementierung eines Interfaces verwechselt werden.
Der springende Punkt ist jetzt die Verbindung des Analyse- mit dem Designmodell. Ein *Function Block* wird auf ein *Package* abgebildet. Andersrum ausgedrückt realisiert (≪*realizes*≫) ein *Package* einen *Function Block*. Eine Akteurklasse (*Actorclass*) realisiert auf gleiche Weise einen Akteur. Beachten Sie, dass es sich hier nicht um 1:1-Beziehungen handeln muss. Anders bei einer Methode und einem Use-Case: Hier stellt genau eine Methode einen Use-Case dar (≪*represents*≫). Die ≪represents≫-Beziehung bezieht sich somit auf eine Darstellung und die ≪realize≫-Beziehung auf eine Realisierung. Später werden wir noch sehen, dass nach dem gleichen Prinzip, sprich mit Hilfe der Metamodellierung, UML Diagramme in Quellcode transformiert werden können. Die Angabe der Kardinalitäten erfolgt auch im Metamodell über die 1-zu-1, 1-zu-*, *-zu-* Notation.

[4] Auf Metamodell-Ebene sind reflexive Beziehungen erlaubt, im Use-Case-Diagramm selbst allerdings nicht.

Wie wir bis jetzt gesehen haben, bestehen zwischen unterschiedlichen Diagrammen auf Metamodellebene Zusammenhänge. Abschnitt 2.4 soll nun die Transformationen eines Klassendiagramms (Abschnitt 2.2) in Programmiercode beispielhaft erklären.

### 2.4 Codegenerierung aus UML Diagrammen

Ziel jeder Modellierung ist irgendwann die Umsetzung des Modells in Programmiercode. In Bezug auf die UML versteht man unter Codegenerierung die Überführung eines UML-Modells in Programmiercode einer gewissen Zielsprache mittels Transformationsregeln. Diese Transformationsregeln ergeben sich, ähnlich wie bei der Transformation des Analysemodells in ein Designmodell, auf Metamodellebene. Auf dieser höheren Metaebene gibt es also Zusammenhänge, die eine eindeutige Transformation eines UML-Diagramms in Programmiercode ermöglichen.

Die Generierung aus einem Modell, steigert nicht zuletzt auch erheblich die Software-Qualität. So werden, z.B. Copy&Paste-Fehler bei der Implementierung ähnlicher Methoden vermieden, da der Programmiercode immer neu generiert wird. Dass solche Copy&Paste-Fehler teuer werden können, hat die Vergangenheit gezeigt. Prominentestes Beispiel ist hier sicherlich der Absturz der Ariane 5 Rakete, bei der es durch die Wiederverwendung von Programmiercode, einer langsameren Vorgängerrakete, zu einem Zahlenüberlauf einer Geschwindigkeitsvariable kam. Dies hatte zur Folge, dass die Rakete explodierte. Der Schaden betrug 850 Mio. Euro.

Obwohl die angesprochenen Transformationen per Hand realisierbar wären, bietet sich hier der Gebrauch von geeigneten CASE-Tools an [7] [8]. In der Praxis werden deshalb zur Modellierung und anschliessenden Transformation in Programmiercode zum grössten Teil nur noch solche Tools eingesetzt. Das Zeichnen der Diagramme und die manuelle Transformation in Programmiercode, würde die Zeit die man duch die Codegenerierung aus Diagrammen gewinnt, wieder zunichte machen. Man könnte so auch Konsistenzfehler durch menschliche Fehler (z.B. Abschreibfehler) nicht mehr ausschliessen.

An einem konkreten Beispiel (Abbildung 6), soll hier die Transformation eines Klassendiagramms in Java-Code gezeigt werden und auf die Probleme, die auftreten können, eingegangen werden.

**Beispiel:** Bei allen „Akteuren“ handelt es sich um eine *Person*, die einen Namen (*name*) und ein Alter (*alter*) hat und die die Methode *trinke(int c)* besitzt. Bei *Person* handelt es sich um eine abstrakte Klasse (*Kursivschrift*) von der keine Instanzen erzeugt werden können.

Sowohl ein *Partyteilnehmer*, also auch ein *Partyveranstalter* sind Personen und erben somit die Attribute und Methoden der Klasse *Person*. Desweiteren enthalten aber auch sie selbst wieder eigene Attribute und Methoden. Sie sind

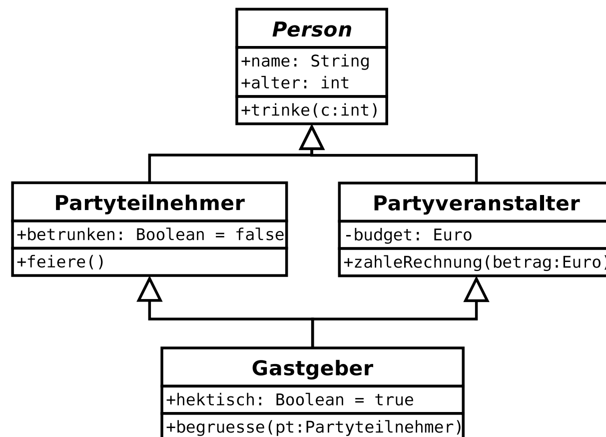


**Abbildung 6.** Klassendiagramm einer Partybeziehung

```
public abstract class Person {
        // Eigenschaften
        public String name;
        public int alter;

        public void trinke(int c) {
                // Programmiercode
        }
}
class Partyteilnehmer extends Person {
        // Eigenschaften
        public Boolean betrunken = false;
        public void feiere() {
                // Programmiercode
        }
 }
class Partyveranstalter extends Person {
        // Eigenschaften
        private int budget;
        public void zahleRechnung(Euro betrag) {
                //Programmiercode
        }
 }
 class Gastgeber ...
         //Mehrfachvererbung ist in Java nicht darstellbar
```

**Abbildung 7.** Java-Code der modellierten Partybeziehung

nicht abstrakt und somit können Instanzen von ihnen erstellt werden. Der *Gastgeber* wiederum ist sowohl *Partyteilneher* als auch *Partyveranstalter* und erbt somit von beiden Superklassen.

Die Umsetzung in Java-Code (Abbildung 7) erfolgt dann entweder per Hand durch Anwenden gewisser Transformationsregeln oder viel komfortabler und praxisrelevanter durch ein UML-Tool welches diese Regeln implementiert und Codegeneration unterstützt (z.B Poseidon for UML[7]). Genau wie beim vorherigen Beispiel der Transformation eines Analyse in ein Designmodell, handelt es sich hier auch um die Transformation eines Modells (UML) in ein anderes Modell (Java-Code). Auf Metaebene bestehen auch hier wieder Zusammenhänge zwischen den beiden Modellen, auf die wir in diesem Beispiel jedoch nicht näher eingehen wollen.

Das in unserem Beispiel modellierte Szenario ist nur bedingt in Java umsetzbar. Da Mehrfachvererbung in Java nicht möglich ist, kann die Klasse Gastgeber nicht in Sourcecode umgesetzt werden. Bei einer anderen Programmiersprache wie C++, die Mehrfachvererbung unterstützt wäre das kein Problem.

Im weiteren Verlauf wollen wir nun darauf eingehen, dass Modelle existieren, die unabhängig von jeglicher Technik sind, und wie man diese plattformunabhängige Modelle, in plattformspezifische Modelle transformieren kann. Abschnitt 2.6 soll dann die Grenzen und Probleme bei der Transformation von Modellen erklären.

## 2.5 Vom Platform Independent Model (PIM) zum Platform Specific Model (PSM)

An dieser Stelle kristalisiert sich ein weiterer Ansatzpunkt für effektivere Entwicklung von Softwaresystemen in Bezug auf die Umsetzung von Modellen in Programmiercode. In vielen Projekten ist die Problemlösung oftmals gleich, nur die Plattform für die die Umsetzung realisiert werden soll ist veränderlich. (Man stelle sich, z.B. einen Algorithmus und seine Implementierung in verschiedenen Programmiersprachen und auf verschiedenen Plattformen (Server, Workstation, Embedded Bereich) vor.) Es ist klar, dass in der Informatik sowohl die Hardwareplattformen, als auch die Softwareplattformen kurzlebig und in ständigem Ausbau und Wandel sind. Es macht also Sinn zuerst plattformunabhängige Modelle (Platform Independent Model(s) - PIM) zu erstellen und diese dann zu einem späteren Zeitpunkt an die jeweilige Plattform anzupassen.
Das PIM ist also ein Modell, das bewusst von technologischen Details abstrahiert. Das Plattform Specific Model (PSM) verwendet die Konzepte einer Plattform um ein System zu beschreiben. Neben dem PIM und dem PSM gibt es im MDA-Prozess auch noch das Computation Independent Model (CIM). Das CIM stellt ein „Businessmodell“ dar, welches komplett unabhängig von irgendwelchen IT-Systemen ist. Wir wollen uns hier aber nur auf den Gebrauch vom PIM und dem PSM beschränken.

Ein klarer Vorteil des PIM liegt darin, dass es auch nach einem Technologiewechsel gültig bleibt. Die klare Trennung der Abstraktionsebenen, und der Fakt, dass Konzepte im allgemeinen stabiler sind als Technologien, garantiert

die Langlebigkeit der Modelle und spart Arbeit und somit Zeit und Geld bei der erneuten Implementierung eines bewährten Modells. Diese Trennung wird somit auch dem Model Driven Architecture (MDA) Konzept der OMG gerecht[5]. Abbildung 8 zeigt das Mapping vom PIM zum PSM. Zum PIM müssen zusätzliche Informationen über die Plattform, an die es angepasst werden soll, vorhanden sein. Diese zusätzlichen Elemente werden dann in das Modell integriert und vereinen somit das PIM mit plattformspezifischen Gegebenheiten. Als Beispiele für solche Plattformen kann man im eBusiness-Bereich CORBA, EJB oder WebServices nennen. Die Programmiersprache C++ wäre dann wiederum eine Plattform für CORBA.

Das PSM muss noch nicht alle Implementierungsdetails enthalten, da es ein Zwischenschritt im MDA-Prozess sein kann, der auf einer gewissen Abstraktionsebene steht und den Ausgangspunkt für weitere Transformationen ist. Am Ende solcher Transaktionsketten steht das PSM dann auf gleicher Modellebene wie der Programmiercode und kann somit auf diesen abgebildet werden.

Im nächsten Abschnitt wollen wir uns nun mit den Grenzen und Problemen der Transformation von Modellen und der Generierung von Programmiercode beschäftigen.

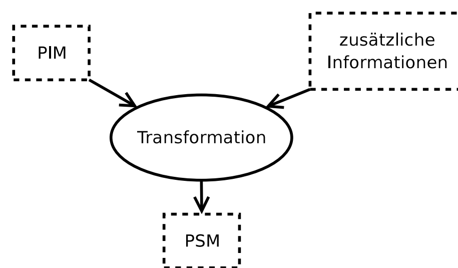


**Abbildung 8.** Umwandlung vom Platform Independant Model (PIM) zum Platform Specific Model (PSM)

### 2.6 Grenzen und Probleme

Obwohl die Möglichkeiten der vorgesehenen Codegenerierung konsequent verbessert wurden, ist auch in der offiziellen UML 2.0-Spezifikation nicht genau angegeben, wie ein UML Tool die Diagramme in Code zu transformieren hat. Obwohl es noch keine konkrete OMG-Spezifikation für eine Transformationssprache auf Basis von Metamodellen gibt, liegt aber schon ein Request for Proposal vor.

Es ist vor allem aber auch nicht möglich und nicht beabsichtigt alle Diagramme

[5] Ein Grundsatz der MDA besteht darin die technologieunabhängigen und die technologiebezogenen Notationen und Konzepte voneinander zu trennen.

(z.B ein Kompositionsstrukturdiagramm) in Programmiercode zu transformieren. Ein gutes UML Tool [7] [8] zeichnet sich aber dadurch aus, wie genau es das jeweilige Modell in Programmiercode transformieren kann und in wieviele Programmiersprachen diese Umwandlung unterstützt wird. Die Transformation vom PIM ins PSM und vom PSM in Quellcode bzw. die umgekehrte Richtung stellt sich als schwierig heraus, ist noch Teil der Forschung und wird aktuell nur von sehr wenigen Tools ansatzweise unterstützt. Eine weitere wichtige Anforderung an ein UML-Tool ist eigene Mappings (PIM → PSM → Quellcode und evtl Quellcode → PSM → PIM) mittels einer formalen Sprache selbst definieren zu können. Entwickler haben somit die Möglichkeit für ihr Projekt spezifische Erweiterungen an der UML vorzunehmen, um somit performantere Mappings für ihr ganz konkretes System zu verwenden. Der Formalismus der Regeln ist insofern wichtig, dass ein Automat eindeutige Transformationen ausführen kann.

### 2.7 Schlussfolgerung

Seit der ersten Annahme der UML als Modellierungssprache seitens der OMG ging es für den Einsatz der UML steil bergauf. Von einigen als „lingua franca“ des Softwareengineerings bezeichnet stellt die UML eine gelungene Zusammenführung praxisrelevanter Techniken dar. Sie konnte sich in der schnelllebigen Softwarebranche etablieren und stellt nun in der Version 2.0 „die“ übliche Modellierungssprache des Softwareengineerings dar. In dieser Version wurden dann auch die Möglichkeiten zur Modelltransformation verbessert, ausführbare Modelle werden teilweise unterstützt und der Modellaustausch zwischen verschiedenen Entwicklungswerkzeugen wurde durch den Gebrauch von XMI-Format verbessert. Auch bei der Geschäftsprozessmodellierung, der Modellierung von Abläufen und dem Echtzeitbereich würden erhebliche Fortschritte gemacht.

Als Softwareentwickler muss man sich allerdings im Klaren sein, dass die UML keine Methode ist, d.h. nicht vorgibt wie der Entwicklungsprozess abzulaufen hat. Sie stellt lediglich einen Satz von Notationen zur Formulierung aus einer allgemeinen Sprache dar.

Obwohl die UML in sehr vielen Bereichen der Softwareentwicklung Verwendung findet, wird sie ihrer Anforderung eine Sprache für alles zu sein (*unified*), nicht ganz gerecht. Genau aus diesem Grund wird die UML ständig weiterentwickelt und es entstehen von der OMG unabhängige Erweiterungen. Zwei Beispiele für solche Erweiterungen sind: UML-RT welche die UML um Echtzeitfähigkeit erweitert, und xUML mit der sich ausführbare Diagramme beschreiben lassen und somit schon Tests in der Planungsphase des Entwicklungsprozesses ermöglichen. Hier sieht man also, dass noch viel Handlungsbedarf besteht, weil die Mittel die zur Verfügung stehen noch nicht komplett ausgereift sind.

Wie wir sehen ist die UML nicht mehr aus dem Softwareengineering wegzudenken und wird in den nächsten Jahren sicherlich noch interessante und praktische Erweiterungen erleben und neue Anwendungsbereiche erschliesse. Die UML stellt schon heute, in der aktuellen Version 2.0 ein wichtiges Werkzeug auf dem Weg zur „komplett“ modellbasierten und generativen Softwareentwicklung dar.

Unter dem Strich, wird sich der Softwareentwicklungsprozess immer mehr in Richtung modellbasierte Entwicklung bewegen. Heute klingt der heißersehnte Wunsch aus einem beliebigen UML Modell, per Knopfdruck, fertige Software zu erstellen (noch) utopisch. Klar ist, dass eine solche Umwälzung in der Softwarebranche viele Programmierer überflüssig machen wird, und Modellierer ins Zentrum des Softwarentwicklungsprozesses stellen wird.

## Literatur


1. **UML Resource Page** - http://www.uml.org/
2. **Object Management Group** - http://www.omg.org/
3. M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins: **UML 2 glasklar** - Hanser Verlag, 2004. (www.uml-glasklar.de)
4. Bernd Oesterreich: **Analyse und Design mit UML 2 Objektorientierte Softwareentwicklung** - Oldenbourg, 2005
5. M. Born, E. Holz, O. Kath: **Softwareentwicklung mit UML 2** - Addison-Wesley, 2004. (www.uml2-buch.de)
6. M. Jeckle, C. Rupp, B. Zengler, S. Queins, J. Hahn: **UML 2.0 Neue Möglichkeiten und alte Probleme** - Informatik Spektrum, August 2004
7. Gentleware: **Poseidon for UML** - http://www.gentleware.com/
8. Telelogic: **Tau/Developer** - http://www.telelogic.com/