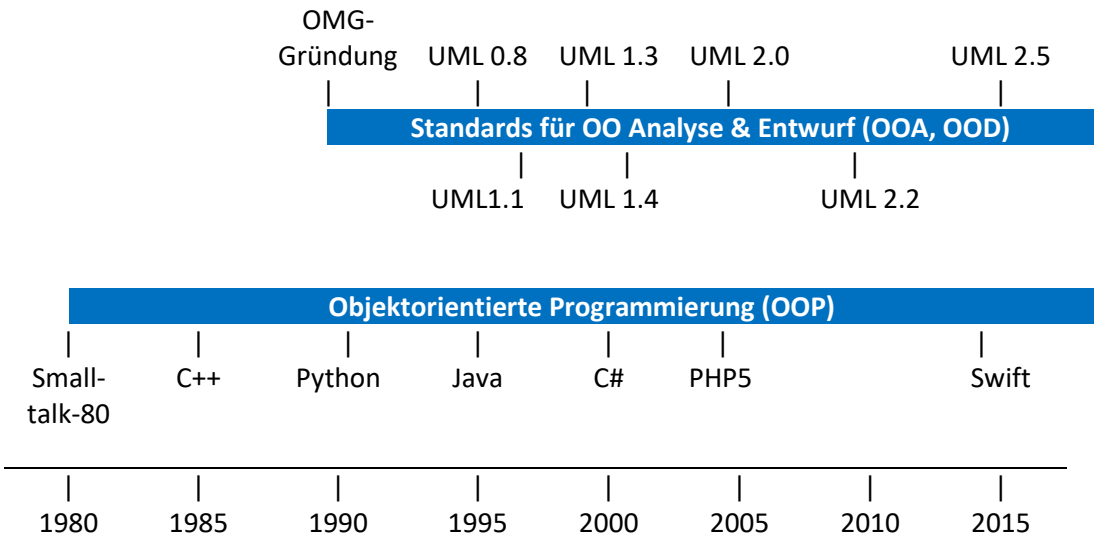
UML Einführung

Wir lernten bereits eine sehr häufige und wichtige Diagramm-Art der Unified Modeling Language (UML) kennen: Das UML Klassendiagramm. Nun gehen wir kurz einen Schritt zurück und überlegen uns, was die UML überhaupt ist und welche Konzepte dahinterstehen. Die Erklärungen basieren auf dem Buch *Lehrbuch der Objektmodellierung* von Heide Balzert (2011).

UML 2.5 von OMG

Bei der UML handelt es sich um eine grafische **Modellierungssprache** zur Spezifikation, Konstruktion und Dokumentation von Software. Sie wird von der Object Management Group (OMG) entwickelt und seit 1995 laufend erweitert. Die derzeit aktuelle Version ist UML 2.5. Folgendes Diagramm gibt einen Überblick über die Entwicklung objektorientierter Programmiersprachen und der UML:



Notation und Semantik

Die UML definiert eine einheitliche Notation und Semantik für ihre Diagramme. Die *Notation* legt die Grafiken (z.B. Klassendiagramm) und Texte (z.B. Spezifikation) fest. Die *Semantik* definiert die Bedeutung der Notation.

Strukturdiagramme

Im Rahmen des UML-Standards wurden 14 Diagramm-Arten definiert. Diese lassen sich in Struktur- und Verhaltensdiagramme unterteilen. *Strukturdiagramme* stellen das **statische Modell** des Systems dar. Denkt man beispielsweise an ein Auto, könnte man im statischen Modell festlegen, welche Eigenschaften ein Auto besitzt (z.B. Geschwindigkeit) und woraus ein Auto besteht: Ein Motor, vier Reifen, Bremsklötze etc. Zum statischen Modell gehören insbesondere Klassen inklusive ihrer Beziehungen zueinander, Objekte und Pakete.

Verhaltens-diagramme

Verhaltensdiagramme zeigen Funktionsabläufe und bilden somit das **dynamische Modell**. Im dynamischen Modell könnte man beispielsweise festlegen, was passiert, wenn das Auto bremst (die Geschwindigkeit wird reduziert etc.). Das dynamische Modell modelliert die wichtigsten Aufgaben und zeigt, wie Objekte miteinander kommunizieren. Zeitliche Abläufe werden ebenfalls abgebildet.

Folgende Tabelle gibt einen Überblick über die 14 Diagramm-Arten:

Die 14 UML Diagramm-Arten

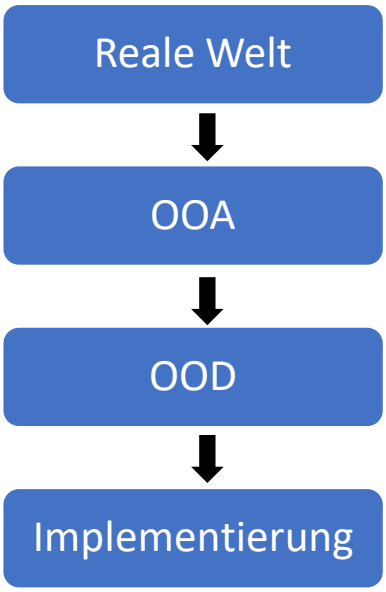
Strukturdiagramme	Verhaltensdiagramme
Klassendiagramm	**Aktivitätsdiagramm**
Objektdiagramm	**Use-Case-Diagramm (Anwendungsfalldiagramm)**
Paketdiagramm	**Zustandsdiagramm**
Kompositionsstrukturdiagramm	**Sequenzdiagramm**
Komponentendiagramm	Interaktionsübersichtsdiagramm
Verteilungsdiagramm	Zeitverlaufsdiagramm
Profildiagramm	Kommunikationsdiagramm

Die hervorgehobenen Diagramm-Arten werden wir uns genauer ansehen.

OOA und OOD

Man unterscheidet zwischen der objektorientierten Analyse (**OOA**) und dem objektorientierten Design (**OOD**), auch objektorientierter Entwurf genannt. Folgendes Diagramm veranschaulicht den Zusammenhang zwischen Analyse, Design und Implementierung:

Von der Realität zur Implementierung



- Anforderungen
- Lastenheft
- Pflichtenheft
- Prototypen
- GUI-Entwurf, Bibliotheken, Datenbank, ...
- Programm (z.B. Java)

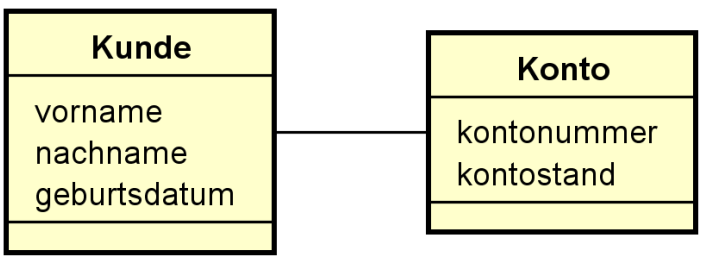
Das Ziel ist es, die komplexe Realität mithilfe von abstrakten Modellen zu vereinfachen. Die UML selbst gibt kein konkretes Vorgehensmodell zur Softwareentwicklung vor. Dies wird beispielsweise im Unified Process definiert. Die bekannteste Realisierung davon ist beispielsweise der Rational Unified Process (RUP) von IBM.

Objektorientierte Analyse

Analysephase

In der Analysephase ermittelt man die Anforderungen des Auftraggebers und modelliert das Fachkonzept. Man denkt hier also noch **nicht an die konkrete Implementierung** (Programmierung)! Alle technischen Aspekte, wie zum Beispiel Programmiersprache, benötigter Speicherplatz, Performance etc., werden hier noch zunächst ausgeblendet. In dieser Phase erstellt man vereinfachte, sehr grobe Diagramme. Folgendes Beispiel zeigt ein solches grobes Klassendiagramm:

Klassendiagramm in der Analysephase



Projekt-dokumentation

Diese Diagramme eignen sich gut für die Kommunikation mit dem Auftraggeber, da sie noch keine detaillierten technischen Informationen wie Datentypen und Sichtbarkeiten beinhalten. Außerdem kann man sie auch für die Projektdokumentation nutzen

und sie ins **Pflichtenheft** oder in die **Machbarkeitsstudie** einfügen. In der Analysephase entsteht typischerweise das Pflichtenheft.

Prototypen

Wir können uns als Menschen schlecht in hypothetische Situationen hineinversetzen. Deshalb ist es bei Software sehr nützlich, *Prototypen* anzufertigen. Dadurch haben Kunden „etwas zum Anfassen“: Gibt es schon eine grobe Skizze der GUI, sehen Kunden, in welche Richtung das Interface geht. Dadurch fällt ihnen beispielsweise ein, dass sie auch noch ein Datumsfeld in einem Formular benötigen. Bemerkt man erst gegen Ende des Projekts, dass ein Datumsfeld fehlt, entstehen viel höhere Kosten: Es müssen nämlich der Code, die Dokumentation, die Tests, das Benutzerhandbuch, die GUI, die Datenbank etc. geändert werden.

Prototyping Tools

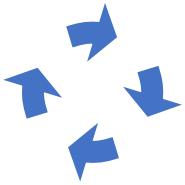
Geeignete Prototyping-Werkzeuge sind all jene Tools, mit denen der/die EntwicklerIn schnell GUI-Skizzen anfertigen kann. Dies reicht von Stift & Papier über PowerPoint zu Prototyping-Tools wie **Balsamiq Mockups**. Auch Prototypen eignen sich gut dafür, im Lasten- und Pflichtenheft zur Dokumentation abgelegt zu werden. Folgender Screenshot zeigt einen einfachen Prototyp in Balsamiq Mockups:

Balsamiq Mockups Prototyp

Wir haben eine Kooperation mit Balsamiq Mockups. Falls du für dein ITP-Projekt eine kostenlose Lizenz haben möchtest, melde dich bei deiner Lehrperson!

Iterationen

Wichtig ist, dass mehrere *Iterationen* durchgeführt werden. Kein Modell ist auf Anhieb vollständig und korrekt. Daher erstellt man die verschiedenen Modelle in mehreren Schritten und arbeitet neue Erkenntnisse in die neuen Versionen der Modelle ein. Wünscht sich nun ein Kunde weitere Felder oder Funktionen, so werden die Diagramme und Prototypen einfach entsprechend angepasst.



Objektorientierter Entwurf

Entwurfsphase

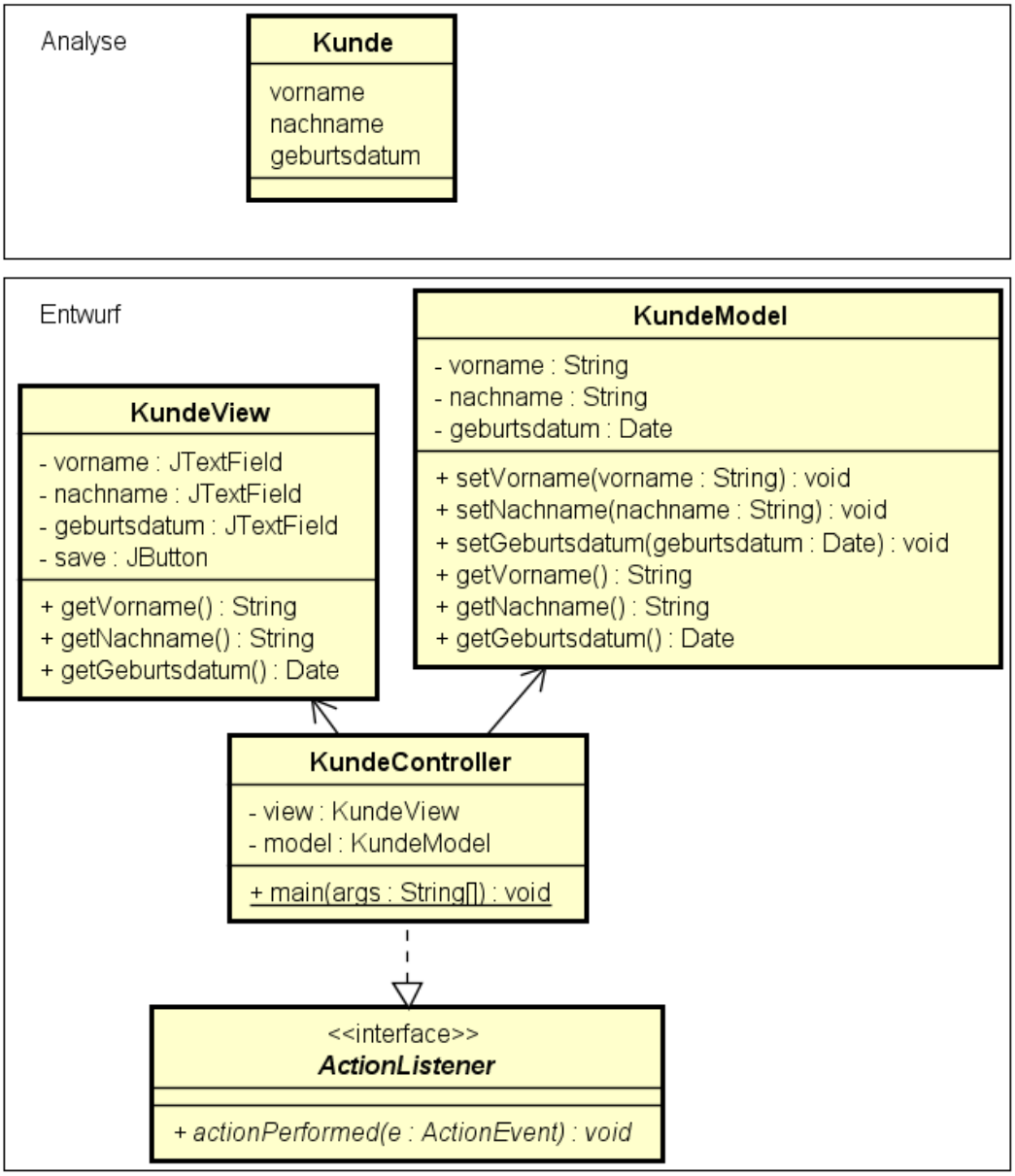
Sobald die Anforderungen stabil sind und sich kaum verändern, kann in die Entwurfsphase übergegangen werden. Spätestens ab diesem Zeitpunkt ist das Projekt normalerweise auf Basis des Pflichtenhefts beauftragt worden. D.h. es wurde meistens ein **Vertrag** unterschrieben, dass man die Software so entwickelt, wie sie im Pflichtenheft spezifiziert wurde.

Datentypen, Sichtbarkeiten, Rückgabewerte, Navigierbarkeit in Entwurf

In der Entwurfsphase arbeitet man grundsätzlich mit denselben Diagramm-Arten wie in der Analysephase. Diesmal spezifiziert man sie jedoch genauer: Man überlegt sich Datentypen, Sichtbarkeiten Rückgabewerte und die Navigierbarkeit (Pfeil-Richtungen von Assoziationen). Außerdem müssen eventuell weitere Klassen hinzugefügt werden, weil man sich beispielsweise für das MVC-Design Pattern entschieden hat. Hier überlegt man sich auch, welche Entwurfsmuster (Design Patterns) für die Aufgabenstellung Sinn machen.

Folgendes Bild verdeutlicht den Unterschied zwischen der Analyse- und Entwurfsphase:

Klasse „Kunde“ in der Analyse- und Entwurfsphase



UML Grundkonzepte

Entwicklung der objektorientierten Programmierung (OOP)

Die Prinzipien der objektorientierten Programmierung haben ihre Wurzeln in Smalltalk-80, der ersten objektorientierten Programmiersprache. Mit der Veröffentlichung von C++ zu Beginn der 90er Jahren wurde das Konzept der objektorientierten Programmierung weiter verbreitet. Daraufhin folgten weitere Programmiersprachen wie Python, Java, C#, PHP5 und Swift, welche den Prinzipien der Objektorientierung folgen.

Die vier Prinzipien der OOP

Die vier wichtigsten **Prinzipien** der objektorientierten Programmierung sind *Abstraktion*, *Datenkapselung*, *Vererbung* und *Polymorphie*.

Abstraktion (1. Prinzip)

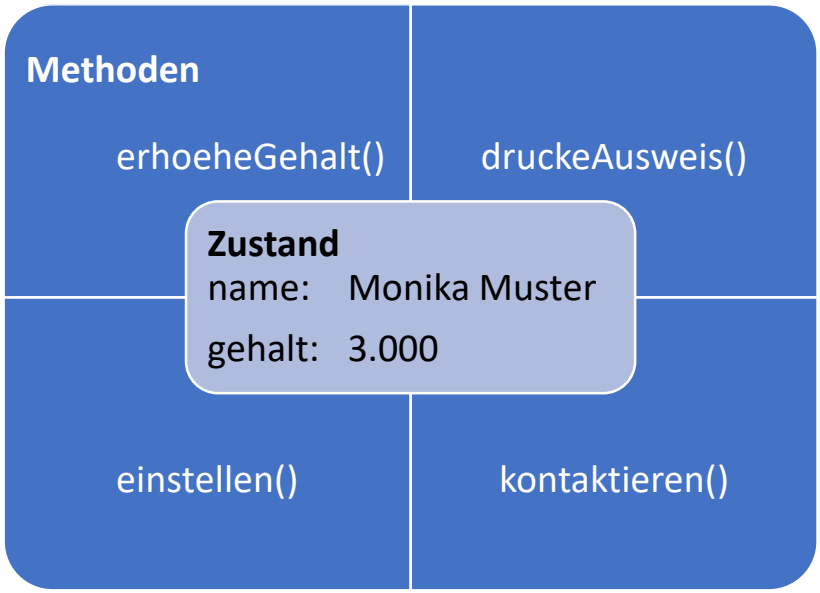
Abstraktion (1. Prinzip) bedeutet, dass wir Objekte der Realität nachbilden. Dabei **beschränken** wir uns jedoch auf die für uns **wichtigsten Informationen** bzw. Aspekte. Welche Aspekte dabei wichtig sind, hängt von der Aufgabenstellung ab. Programmieren wir beispielsweise eine Schülerverwaltung, könnten Vor- und Nachname, Adresse, Telefonnummer, E-Mail-Adresse und Sozialversicherungsnummer relevant sein. Die Schuhgröße ist jedoch in diesem Falle uninteressant, weshalb wir diese nicht abbilden müssen. Entwerfen wir jedoch keine Schülerverwaltung, sondern eine Kundendatenbank für einen Schuhladen, könnte die Schuhgröße wiederum sehr wohl interessant sein. Wir *abstrahieren* also, indem wir nur die wichtigsten Informationen modellieren. Wir fassen nun dieses Abbild der Realität in Klassen zusammen. Es ist wichtig, zwischen Klassen und Objekten zu unterscheiden. Eine Klasse legt für seine Objekte fest, wie die *Struktur* (Attribute), das *Verhalten* (Operationen bzw. Methoden) und die *Beziehungen* (Assoziationen) auszusehen haben.

Datenkapselung (2. Prinzip)

Jedes Objekt einer Klasse hat einen konkreten Zustand (Attributwerte und Objektbeziehungen). Der **Zustand** darf im Sinne der *Datenkapselung* (2. Prinzip) von außen **nur durch Methoden geändert** und abgefragt werden, also durch das Verhalten. Dies wird auch Geheimnisprinzip genannt. Man stellt sich auch oft ein Modell des „Nachrichtenaustauschs“ vor, d.h. ein Objekt empfängt Nachrichten, ändert ggf. seinen Zustand und antwortet möglicherweise mit einer Nachricht. Das ist nichts anderes als der Aufruf von Methoden mit Parametern inkl. Rückgabewert.

Folgendes Beispiel zeigt den schematischen Aufbau eines `Mitarbeiter`-Objekts:

Datenkapselung: Objekt Mitarbeiter



Man sieht, dass der innere Zustand des `Mitarbeiter`s, nämlich der `name` und das `gehalt`, „geschützt" ist. Sie können von außen nicht direkt verändert werden. Dies kann ausschließlich über das Verhalten geschehen, also über die Methoden `erhoeheGehalt()`, `druckeAusweis()`, `einstellen()` und `kontaktieren()`. In Java löst man dies, indem Attribute standardmäßig `private` und Operationen `public` sind.

Setter, Getter und Datenkapselung

Warum sollte man sich überhaupt an das Geheimnisprinzip halten, wenn Getter und Setter ja erst wieder den Zugriff ermöglichen?

Manchmal machen Getter und Setter tatsächlich nichts Anderes als einen „direkten" Zugriff zu ermöglichen. Oftmals müssen jedoch zusätzliche Überprüfungen durchgeführt werden: Bei einer Klasse `Person` könnte beispielsweise in `setAlter(int alter)` überprüft werden, ob die Zahl positiv ist. Getter und Setter garantieren daher einen **ordnungsgemäßen Zugriff** auf die Attribute.

Vererbung (3. Prinzip)

Auch die *Vererbung* (3. Prinzip) haben wir bereits kennengelernt. Sie beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse, Superklasse, Oberklasse) und einer spezialisierten Klasse (Subklasse, Unterklasse). Ein Objekt der spezialisierten Klasse kann überall dort eingesetzt werden, wo auch ein Objekt der allgemeinen Klasse erlaubt ist. Es besteht daher eine **ist-ein-Beziehung**: Jedes

Objekt der Subklasse *ist ein* Objekt der Superklasse. Details zur Vererbung finden sich im folgenden Kapitel UML Klassendiagramm.

Polymorphie (4. Prinzip)

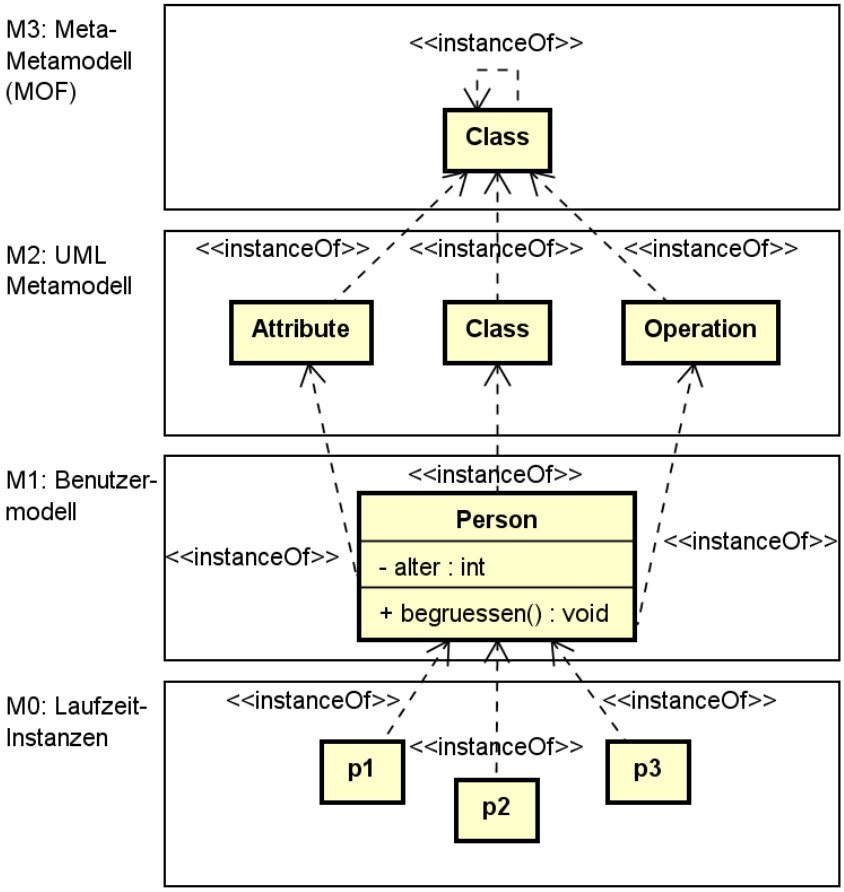
Auch die *Polymorphie* (4. Prinzip) haben wir bereits kennengelernt und ausführlich besprochen. Die Vererbungspolymorphie (in Java: Laufzeitpolymorphie) ermöglicht, dass erst zur Laufzeit bestimmt wird, welche konkrete Operation (Methode) ausgeführt wird – je nachdem, um welches Objekt es sich tatsächlich handelt. Man spricht auch vom **dynamischen Binden** oder späten Binden. Dieses Konzept wird in unterschiedlichen Programmiersprachen verschieden implementiert. Während in Java alle Objektoperationen automatisch polymorph sind, muss dies in C++ beispielsweise explizit deklariert werden.

Diese Prinzipien der Objektorientierung sind Grundlage für alle Diagrammarten. Zunächst sehen wir uns das Objektdiagramm und das Klassendiagramm an – die zwei wahrscheinlich wichtigsten Strukturdiagramme.

Erweiterte Kompetenzen

Die UML definiert ein sogenanntes *Metamodell*. Dieses beschreibt, wie UML-Diagramme auszusehen haben. Das Metamodell selbst ist wieder ein Modell, dessen Aufbau in einem Meta-Metamodell beschrieben wird. Folgende Grafik veranschaulicht den Aufbau (Hierarchie) der Metamodellierung:

Hierarchie der Metamodellierung und MOF



Die Basis aller Modelle bildet das Meta-Metamodell „Meta Object Facility“ (MOF) auf der obersten Ebene M3. MOF wurde von der OMG definiert, um Gemeinsamkeiten von Modellierungssprachen zu beschreiben. MOF ist so gestaltet, dass es sein eigenes Modell bildet und sich somit selbst an die eigenen Regeln hält. UML befindet sich auf der Ebene M2 und befolgt die Regeln von MOF. Hier wird definiert, welche Elemente ein UML-Diagramm beinhalten darf (z.B. Attribute, Assoziationen etc.). Wenn wir nun ein eigenes UML Diagramm designen, befinden wir uns auf M1. Unsere programmierte Software wird schließlich auf der untersten Ebene M0 ausgeführt. Manchmal wird M0 stattdessen auch als „Realität“, also die modellierten Objekte der Realität, beschrieben. Die Hierarchie der Metamodellierung wird nach oben hin immer abstrakter und allgemeiner.

Habe ich es verstanden?

- Ich kann den Begriff UML und ihre Geschichte erklären
- Ich kenne den Unterschied zwischen Notation und Semantik
- Ich kann zwischen Struktur- und Verhaltensdiagrammen unterscheiden, diese erklären und Beispiele dafür nennen
- Ich kann die Begriffe OOA und OOD sowie deren Zusammenhang mit der Implementierung erklären
- Ich kann die objektorientierte Analyse sowie die eingesetzten Methoden erklären
- Ich kann den objektorientierten Entwurf sowie die eingesetzten Methoden erklären
- Ich kann die vier Prinzipien der objektorientierten Programmierung aufzählen und diese erklären
- (*Erweitert*) Ich kann die Hierarchie der Metamodellierung skizzieren und erklären