C++ FOR
(Dienstagvormittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt im direkten Anschluss an die Mittagspause.

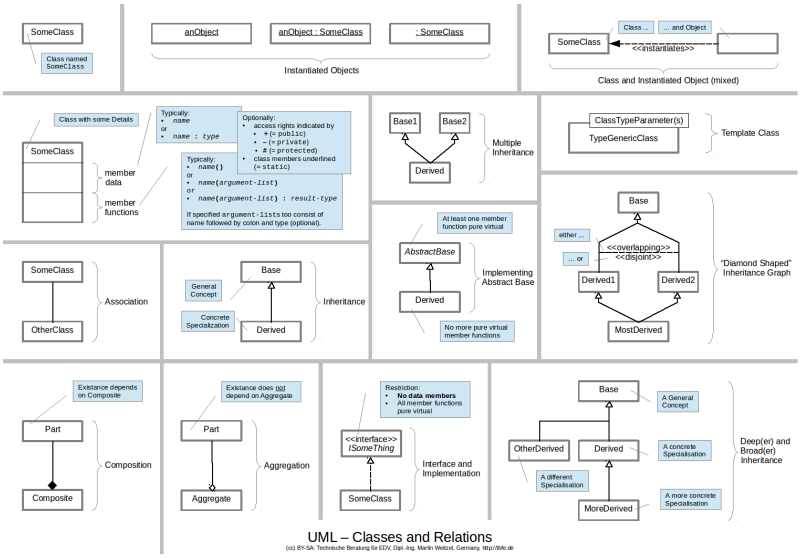
Beziehungen zwischen Klassen

- Klasse (minimal)
- Klasse (detailiert)
- Instanziiertes Objekt
- Parametrisierte Klasse

- Assoziation
- Komposition
- Aggregation

- (Mehrfach-) Vererbung
- Interface
- Abstrakte Basisklasse

- "Rautenförmige" und ...
- ... allgemein mehrstufige Vererbungs-Hierarchien



Click here for Info-Graphic
UML-ClassesAndRelations
Some Class

Klasse (minimal)

Die minmale Darstellung einer Klasse in der [UML] besteht aus einem Rechteck, in welchem der Klassenname steht.

> Klassen und Beziehungen zwischen Klassen sind die wichtigsten Bestandeteile einer *Objekt-Orientierten-Modellierung*.

Häufig wird eine solche etwa

- beginnen mit einem High-Level-Design ([HLD])
- das u.U. eine Reihe von Verfeinerungsschritten durchläuft
- ...
- und schließlich enden mit einer Lösungs-Implementierung.*

*: Auch im Kern sehr formale Prozesse wie etwa RUP beinhalten dabei die Möglichkeit eines "Tailoring", indem nur diejenigen Artefakte tatsächlich erstellt werden, von denen man sich einen konkreten Nutzen verspricht.

Objektorientierte Vorgehensweise

Die übliche Abfolge der Schritte ist

- zunächst eine Objektorientierte (Problem-) Analyse (OOA) vorzunehmen,
- der ein mehr oder weniger detailliertes Objekt-Orientiertes (Lösungs-) Design (OOD) folgt.

Die beiden Begriffe *OOA* und *OOD* werden mitunter auch zusammengefasst zu OOAD (Objektorientierte Analyse und Design).

Klassen vs. Objekte

Hierfür dient in der UML im wesentlichen die selbe Symbolik - der Unterschied besteht lediglich darin, dass

- bei Objekten der Name unterstrichen ist und
- der Klassenname auch entfallen kann*

Klassen

Dies kommen einem **Bauplan** gleich und enthalten in der UML-Darstellung alles, was die Objekten gemeinsam haben, die gemäß diesem Bauplan erstellt werden.

Objekte

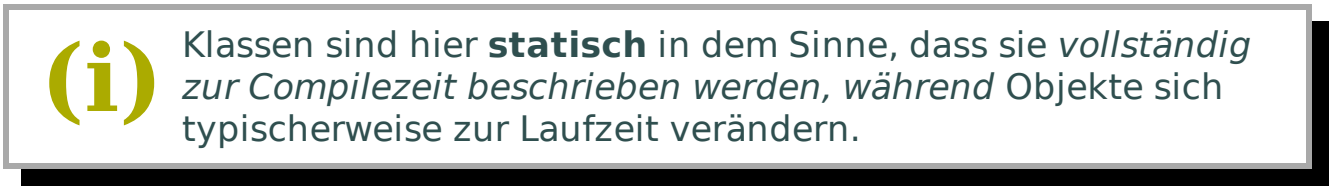
Diese sind *instanzierte Klassen* und drücken in der UML-Darstellung damit die **Unterschiede** aus, welche trotz des gemeinsamen Bauplans bestehen.

---

*: Die tatsächlichen Regeln der grafischen Darstellung sind komplizierter und erlauben prinzipiell, dass ein dem Namen des Objekts ein Doppelpunkt und dann der Klassenname folgt, wobei beide Namen optional sind und weggelassen werden können (beispielsweise wenn sie sich aus dem Kontext ergeben).

Klassen und Objekte in C++

Als kompilierte Sprache gibt es in C++ einen weiteren, charakteristischen Unterschied zwischen Klassen und Objekten:



C++-Klassen als Compilezeit-Konstrukt

Die vollständige Festlegung einer Klasse zur Compilezeit ist ein wesentlicher Unterschied zu deutlich dynamischeren OOP-Sprachen, wie etwa

- Smalltalk oder
- Python

wo **nicht zwingend** Member-Daten und -Funktionen auch noch zur Laufzeit hinzugefügt oder entfernt werden können.*

*: Java und C# sind in dieser Hinsicht wiederum eher ähnlich zu C++, besitzen allerdings mehr standardisierte Möglichkeiten zur Introspektion und können mit Hilfe kleiner Kunstgriffe den Eindruck eines zur Laufzeit dynamisches Verhalten bieten.

Klasse (detailiert)

In stärkerer Detaillierung beschrieben kann einer Klasse im UML-Diagramm folgendes hinzugefügt werden:

- Ein Abschnitt mit Attributen
- Ein Abschnitt mit Methoden

Beides wird innerhalb des Klassensymbols durch eine waagrechte Linie getrennt und beides ist optional.

> Pragmatiker weisen gerne darauf hin, dass die UML **keineswegs** die Nutzung aller ihrer notationellen Möglichkeiten vorschreibt sondern der Detaillierungsgrad eines Klassendiagramm stets an dessen Zweck bzw. dem Zielpublikum ausgerichtet sein sollte!

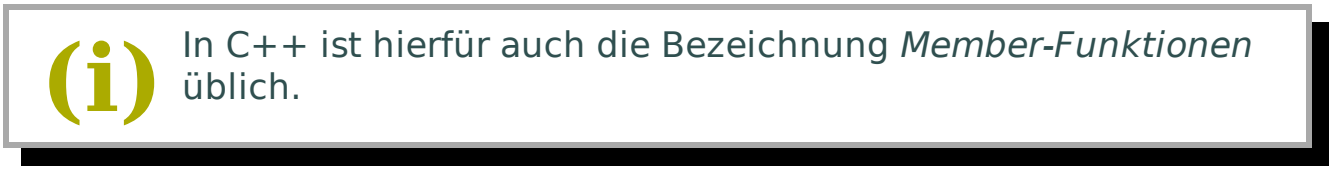
Attribute

Diese enthalten *Datenwerte* und sind im UML-Diagramm daran zu erkennen, dass ihnen **keine** runden Klammern folgen. Ein nachgestellter Datentyp ist dabei optional.

In C++ ist hierfür auch die Bezeichnung *Member-Daten* üblich.

Methoden

Diese enthalten *ausführbare Abläufe* und sind im UML-Diagramm daran zu erkennen, dass ihnen **runde Klammern folgen**, evtl. auch leere.



Innerhalb der Klammern können optional zu übergebende Parameter benannt werden, inklusive deren Typ (durch Doppelpunkt getrennt), und den Klammern folgen kann (ebenfalls durch einen Doppelpunkt getrennt) der Rückgabetyp.

Zugriffsrechte

Ein weiteres, in der detaillierten Darstellung von UML-Klassen mögliches Notationselement bezieht sich auf die Zugriffsrechte und kann sowohl Member-Daten wie auch Member-Funktionen optional vorangestellt werden:

- + (öffentlich) - jeder, der Zugriff zu einem Objekt dieser Klasse hat, kann auf das Attributs zugreifen oder die Methode ausführen;
- # (geschützt) - erreichbar nur für abgeleitete Klassen;
- - (privat) - erreichbar nur in den eigenen Member.

Klassenattribute und -methode

In der detaillierten Darstellung einer UML-Klasse unterstrichene Bezeichner stellen sog. Klassen-Member dar und entsprechen in der C++-Programmierung den Zusatz `static`.

Klassenattribute

Diese sind **nicht** pro Objekt sondern pro Klasse gespeichert*

Klassenmethoden

Diese können ausschließlich auf Klassenattribute zugreifen. Sie bieten in C++ zum einen eine kleine Effinzienzverbesserung, da bei ihrem Aufruf keine Objektadresse übergeben werden muss, und sind zum anderen auch ohne Bezug auf ein bestimmtes Objekt unter Voranstellen des Klassenamens gefolgt vom Scope-Operator (`::`) aufrufbar.

---

* Eine typische Anwendung wäre z.B. eine fortlaufende Nummerierung aller jemals erzeugten Objekte einer bestimmten Klasse, wobei das Klassenattribut die nächste zu vergebende Nummer enthalten könnte.

Parametrisierte Klasse

Hierbei handelt es sich um die UML-Sichtweise auf das, was in C++ mit dem Mechanismus der Templates verfügbar ist und auch die Bezeichnung *generische Programmierung* trägt.

Ursprünglich verdankt diese ihre Entstehung dem Wunsch, Datentypen und Compilezeit-Konstanten wie etwa die Anzahl der Elemente in einem Array parametrisieren zu können.

Assoziation

Eine Assoziation ist die unspezifischste aller Beziehungen, die zwischen zwei Klassen bestehen können und besagt mehr oder weniger nur, dass diese beiden Klassen eine bestimmte Aufgabe **gemeinsam** erledigen.

Die tatsächliche Beziehung kann stärker sein und auch eine der anderen Beziehungsformen annehmen, insbesondere die der

- Aggregation oder
- Komposition.

Ungerichtete Assoziation in C++

Streng genommen bedeutet die allgemeine Assoziation in C++, dass die beiden Klassen gegenseitig aufeinander verweisen, was z.B. in Form zweier Pointer geschehen könnte.

(i) Programmiertechnisch lässt sich nur einer dieser beiden Pointer durch eine Referenz ersetzen, denn bei zwei Referenzen ist die (wechselseitige) Initialisierung nicht realisierbar.

Gerichtete Assoziation

Durch hinzufügen eines Pfeils an der Assoziationslinie lässt sich zum Ausdruck bringen, welche Klasse die andere erreichen kann und welche nur erreichbar ist.

(i) Für die programmiertechnischen Umsetzung kommt damit klar zum Ausdruck, welche Klasse in der Lage sein muss, die jweils andere - z.B. per Zeiger oder Referenz - zu erreichen.*

Ein (bereits existierendes) Objekt erreichbare Klasse kann dann beispielsweise im Konstruktor bei der Erzeugung eines Objekts der anderen Klasse als Argument übergeben und zur Initialisierung eines entsprechenden (Daten-) Members verwendet werden.

*: Die direkte Einbettung der erreichbaren Klasse als Member-Datum entspräche der jedoch der Komposition.

Komposition

Die Komposition ist eine stärkere aber häufig auftretende Form der Aggregation, bei der das Teil in seiner Lebenszeit an die Gesamtheit gebunden ist.

In der UML-Darstellung wird an die (Assoziations-) Linie zwischen den beiden beteiligten Klassen auf der Seite der Gesamtheit eine *ausgefüllte* Raute hinzugefügt.

Aggregation

Die Komposition ist eine geringfügig stärkere Form der Assoziation, indem sie

- die Klasse auf einer Seite zur *Gesamtheit* (Aggregat)
- die Klasse auf anderen zu *deren Teil* erklärt.

Allerdings können Gesamtheit und Teil unabhängig voneiander existieren.

In der UML-Darstellung wird an die (Assoziations-) Linie zwischen den beiden beteiligten Klassen auf der Seite der Gesamtheit eine *nicht-ausgefüllte* Raute hinzugefügt.*

*: Aus pragmatischer Sicht ist darauf hnizuweisen, dass Aggregation und Assoziation häufig eng beieinander liegen und mitunter sogar "Ansichtssache" sind, womit sich ein längeres Nachdenken oder gar eine kontroverse Diskussion darüber, welche Art der Beziehung denn nun per UML darzustellen ist, nicht lohnt.

Vererbung

Hierbei geht es um die Beziehung zwischen einem

- allgemeinen Konzept und
- dessen Spezialisierung.

In der UML-Darstellung wird an die (Assoziations-) Linie zwischen den beiden beteiligten Klassen auf der Seite der Basisklasse ein nicht ausgefülltes Dreieck hinzugefügt.*

Das Speicher-Layout und auch die Lebenszeit-Kopplung entsprechen zwar dem der Komposition, als wichtige besonderheit gilt jedoch das LSP.

(i) Das Liskovsche Ersetzungs-Prinzip ist eng mit dem Gedanken der Objektorientierung verknüft und besagt, dass ein Objekt einer abgeleiteten Klasse (Spezialisierung) stets ein passender Stellvertreter für ein Objekt seiner Basisklasse sein kann.

Mehrfachvererbung

Diese liegt vor, wenn eine Klasse zwei Basisklassen hat, also zwei allgemeine Konzepte spezialisiert.

In der Geschichte der Objektorientierung gab es zahlreiche Diskussionen über Sinn und Zweck der Mehrfachvererbung und in der Tat verzichtet eine Reihe von OOP-Sprachen mit großer Verbreitung - etwa Java - völlig darauf.*

*: Was Java allerdings unterstützt sind Klassen, die mehrere Interfaces implementieren.

Interface (Schnittstelle)

Bei einem Interface geht es prinzipiell um eine Art Vertrag zwischen zwei Klassen:

- Es gibt einerseits eine (oder mehrere) Klasse(n), die ein bestimmtes Interface implementieren und
- für (in der Regel mehrere) andere Klassen ist damit klar, wie das "Angebot" an Member-Funktionen aussieht.

In der UML-Notation steht über dem Namen eines Interfaces das Stereotype «interface».

Anwendung von Interfaces

Generell reduzieren Interfaces die Kopplung, d.h. den Grad zu dem verschiedene Klassen Details voneinander kennen (müssen).

Zudem bieten sie Flexibilität zur Laufzeit, indem von mehreren verfügbaren Implemetierungen die für eine Situation passende gewählt werden kann.

Interfaces haben wor allem auch durch die objektorientierten Entwurfsmustern, die auch zentrales Thema des GoF-Buch sind, eine große Bekanntheit erlangten.

Interfaces vs. Klassen

Technisch gesehen sind Interfaces in C++ in einer bestimmten Form eingeschränkte Klassen:

- Sie besitzen **keine Member-Daten** und
- **ausschließlich rein virtuellen Member-Funktionen**.

Abstrakte Basisklasse

Die Bezeichnung "abstrakt" wird für Klassen verwendet, die **mindestens eine rein virtuelle Member-Funktion** haben.

In der UML-Darstellung wird der Name einer abstrakten Klasse kursiv geschrieben.

Abstrakte Klassen können nur als Basisklassen verwendet werden. Die abgeleitete Klasse wird dann typischerweise die rein virtuellen Member-Funktionen implementieren.

[!] Der Versuch, Objekte einer abstrakten Klasse zu erzeugen scheitert in C++ mit einem Kompilierfehler.

"Rautenförmige" Vererbungsstruktur

Eine solche entsteht, wenn sich bei Mehrfachvererbung auf der mittleren Ebene Klassen befinden, die wiederum eine gemeinsame Basisklasse besitzen.

Es ist bezüglich der gemeinsamen Basisklasse dann in der UML-Darstellung eine Unterscheidung zwischen zwei Fällen erforderlichs, welche jeweils die Daten-Member der (gemeinsamen) Basisklasse betreffen:

- «overlapping» - die Daten-Member sind nur einmal vorhanden
- «disjoint» - die Daten-Member sind doppelt vorhanden

[!] Im zweiten Fall ist in Bezug auf die ganz oben stehende Basisklasse von der ganz unten stehenden Klasse aus gesehen das LSP außer Kraft gesetzt, da keine Eindeutigkeit mehr gegeben ist.

Mehrstufige Vererbung allgemein gesehen

Im Allgemeinen gehen Vererbungs-Hierarchien auch

- *mehr in die Breite*, d.h. eine Basisklasse wird oft viele (direkt) abgeleitete Klassen haben, sowie
- *mehr in die Tiefe*, d.h. es gibt mehrere Stufen (also sieht man die direkt abgeleiteten Klassen als "Kinder", wird es also auch "Enkel", "Ur-Enkel" usw. geben.

Ausgenommen rautenförmigen Hierarchien mit gemeinsamen «disjoint» Basisklassen gilt das LSP automatisch über alle Stufen einer in die Tiefe gehenden Vererbungshierarchie.

Mehrfachvererbung und virtuelle Basisklassen

- Prinzip der Mehrfachvererbung
- Virtuelle Basisklassen

- Überlappende Basisklassen

Prinzip der Mehrfachvererbung

Bei der Mehrfachvererbung kann im Speicherlayout nur noch eine der Basisklassen eine gemeinsame Anfangsadresse mit der abgeleiteten Klasse haben.

Dies stellt keine große technische Herausforderung dar sondern bedeutet lediglich eine minimale Komplikation bei der Umsetzung des LSP, wo im Fall der Weiterreichung der abgeeitete Klasse ggf. ein Offset zum `this`-Zeiger hinzugerechnet werden muss.

Virtuelle Basisklassen

Virtuelle Basisklassen bilden in C++ den Hintergrund der Lösung dessen, was die UML bei rautenförmigen Vererbungshierarchien im Fall von «overlapping»»-Klassen an der Spitze verlangt.

Den Overhead für diese Lösung haben die Klassen auf der mittleren Ebene zu tragen, indem sie nicht nur den Datenteil ihrer (virtuellen) Basisklasse enthalten sondern einen zusätzlichen Zeiger, über den **alle Bezugnahmen auf den Basisklassenteil** erfolgen.

Überlappende Basisklassen

Im konkreten Fall einer rautenförmigen Hierarchie wird nun

- die Daten-Member der Basisklasse nur ein einziges Mal in der "Most Derived Class" vorhanden sein, und
- die Zeiger in den Klassen der mittleren Ebene werden **beide** genau auf diesen Daten-Member verweisen.

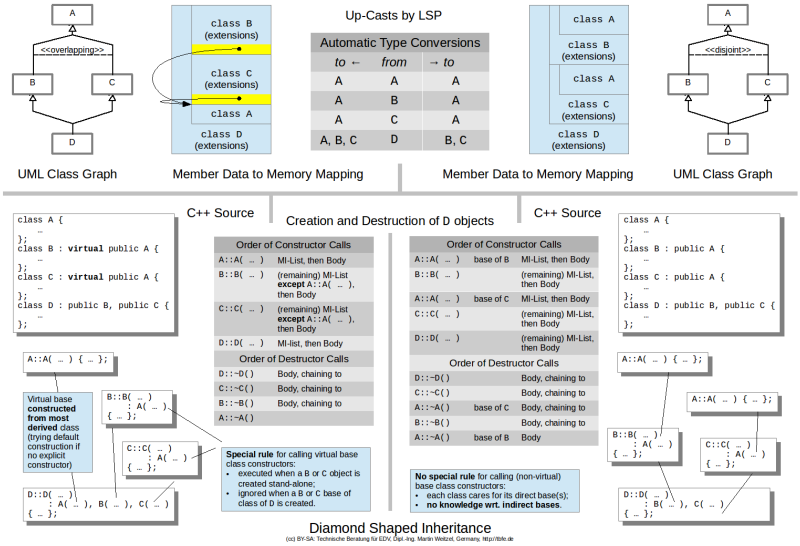
Da alle Bezugnahmen darauf - wie gerade beschrieben - über diese Zeiger laufen, werden die Klassen der mittleren Ebene also mit **ein und demselben** Basisklassen-Objekt arbeiten.

Rautenförmige Ableitungshierarchien

- UML-Darstellung «overlapping» ...
- ... versus «disjoint»

- Abbildung auf Speicher «overlapping» ...
- ... versus «disjoint»

- Konstruktor / Destruktor «overlapping» ...
- ... versus «disjoint»



UML-Darstellung «overlapping»

Die zusammengefasste Darstellung des Falls «overlapping» in Bezug auf

- die UML-Darstellung,
- das Speicherlayouts und
- die Typumwandlungen im Rahmen des LSP

erlaubt den drekten Vergleich zum Fall «disjoint».

UML-Darstellung «disjoint»

Die zusammengefasste Darstellung des Falls «disjoint» in Bezug auf

- die UML-Darstellung,
- das Speicherlayouts und
- die Typumwandlungen im Rahmen des LSP

erlaubt den drekten Vergleich zum Fall «overlapping».

Abbildung auf Speicher «overlapping»

Die zusammengefasste Darstellung des Falls «overlapping» in Bezug auf

- den Quelltext,
- den konstruktor sowie
- den Destruktor-Ablauf

erlaubt den drekten Vergleich zum Fall «disjoint».

Abbildung auf Speicher «disjoint»

Die zusammengefasste Darstellung des Falls «disjoint» in Bezug auf

- den Quelltext,
- den konstruktor sowie
- den Destruktor-Ablauf

erlaubt den drekten Vergleich zum Fall «overlapping».

Beispiele zu Klassenbeziehungen

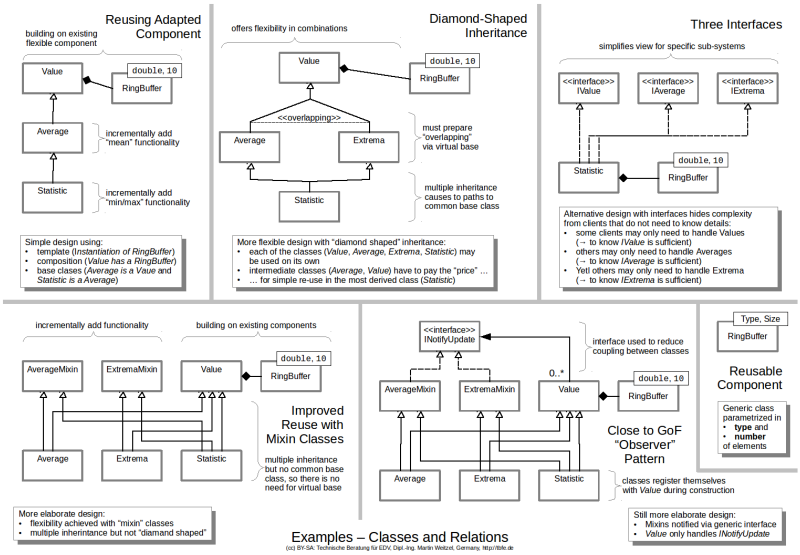
- Eine anpassbare Komponente ...
- ... und deren Nutzung

- Rautenförmige (Mehrfach-) Vererbung

- Drei Interfaces

- Flexible Erweiterbarkeit durch "Mix-Ins"

- Orientiert am GoF "Observer" Muster



Eine (anpassbare) Komponente

Anpassbare Komponenten entstehen normalerweise entweder

- durch Planung, wenn zukünftige Variabilität (korrekt) vorhergesehen wurde, oder
- durch Erfahrung, zu einer bereits bestehende Funktionalität eine (im ersten Entwurf) noch nicht eingeplante neue Variante benötigt wird.

Nutzung der (angepassten) Komponente

Die Nutzung einer (angepassten) Komponente reduziert den Aufwand an immer wieder neu zu schreibendem Code.

Rautenförmige (Mehrfach-) Vererbung

Über den "Preis", der für die dann oft notwendigen, virtuellen Basisklassen zu zahlen ist, lassen sich ggf. flexible Kombinationsmöglichkeiten mit einem guten Grad an Wiederverwendung erreichen.

Drei Interfaces

Die Nutzung von Interfaces kann die Komplexität der Gesamt-Architektur aus der "Sicht einzelner Klienten" etwas vereinfachen.

Inkrementale Erweiterbarkeit durch Mix-Ins

Über Mixin-Klassen lässt sich ggf. flexible Kombinierbarkeit schaffen auch ohne das virtuelle Basisklassen notwendig werden.

Orientiert am GoF "Observer" Muster

Das "Flaggschiff" unter allen hier verglichenen Entwürfen.

Es vereint hohe Flexibilität und geringe Kopplung ... erfordert allerdings auch etwas tiefere Beschäftigung damit, um es endgültig zu verstehen.

Übung

Ziel der Aufgabe:

Analyse und Bewertungen diverser Varianten eines Beispiels, in dem eine Reihe unterschiedlicher Klassenbeziehungen verwendet werden.

> Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.