

Informatik 3

Objektorientierter Entwurf

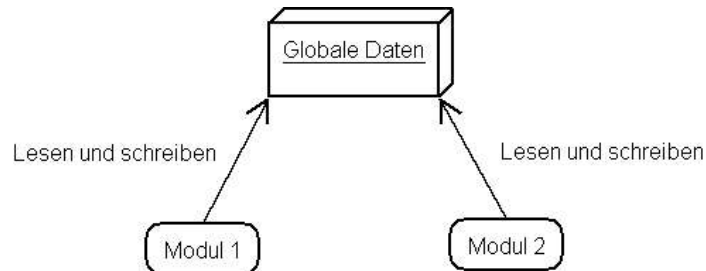
Autor:	Rüdiger Marwein
Datum:	2004-12-27

Aufgabe 3.1 - Globale Daten, Module, Klassen

Szenario a

Zwei Module greifen auf dieselben globalen Daten zu

(1)



(2)

Bei der Spezifikation muss die Konsistenz der globalen Variablen in den verwendenden Modulen abgedeckt sein. Dies kann durch globale Funktionen oder durch vervielfältigte Prüfungen in jedem Modul erreicht werden.

In der Schnittstellendokumentation muss ein dedizierter Hinweis auf den Gebrauch der globalen Daten vorhanden sein, der idealerweise auch die anderen Module benennt, die diese Daten verwenden.

(3)

Die Konsistenz der Programmeinheit ist durch globale Funktionen oder vervielfältigte Prüfungen in jedem Modul erreichbar. Da dies aber auch entfallen lassen werden kann, ist die Konsistenz nur bedingt gewährleistet.

(4)

Die Programmeinheit ist potentiell nur wiederverwendbar wenn ihre Rahmenbedingungen (globale Daten) vorhanden sind. Sie können aufgrund bspw. gleichnamiger bereits vorhandener Variablen nicht beliebig in ein bestehendes System eingesetzt werden.

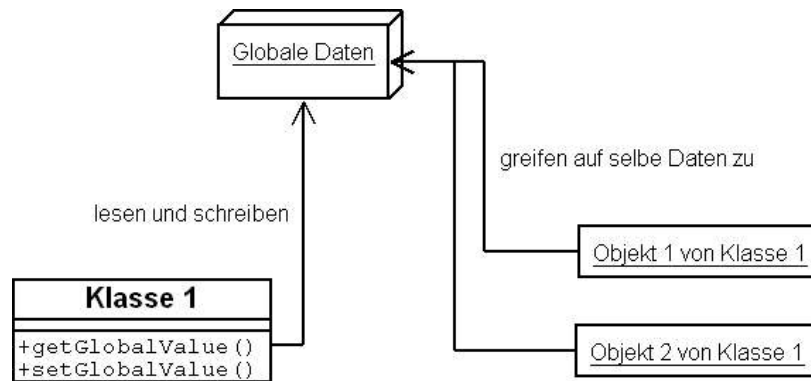
(5)

Diesen Weg zu gehen ist auf kurze Sicht bequem, die Daten liegen in einer Wolke und können nach belieben von jedem benutzt und geändert werden. Eine klare Zuordnung wird nur durch die Dokumentation geschaffen und diese Verknüpfung ist mehr als Zweifelhaft.

Szenario b

Eine Klasse greift auf globale Daten zu, sodass jedes Objekt dieser Klasse auf diese Daten zugreifen kann.

(1)



(2)

Bei der Spezifikation wird dies vermutlich als Klassenvariable auftauchen.

Auch hier muss bei der Dokumentation stets auf die globalen Daten verwiesen werden.

(3)

Die Konsistenz ist von der benutzenden Klasse zu gewährleisten, falls nur diese die Daten verwendet.

(4)

Die Programmeinheiten, die nicht auf globale Daten zugreifen. Siehe a.

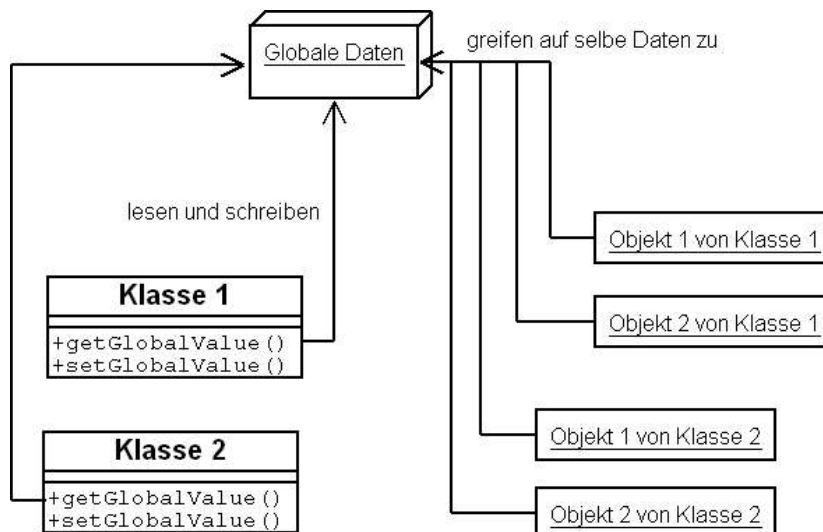
(5)

Solch ein Verhalten wird durch Klassenvariablen erreicht. Bei Programmiersprachen, die so etwas nicht kennen, kann man diesen Weg gehen, muss es aber genauestens Dokumentieren, falls diese Variable ausserhalb ebenfalls übergreifend gültig sein soll.

Szenario c

Zwei Klassen greifen auf dieselben globalen Daten zu, sodass Objekte verschiedener Klassen auf diese Daten zugreifen können.

(1)



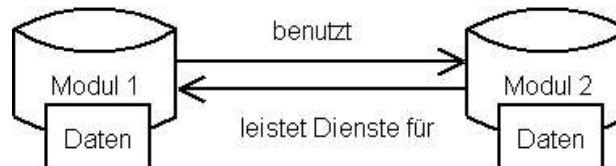
(2)-(4)

Das ist im Prinzip das gleiche wie a, abgesehen, davon, dass mehr als eine Instanz jeder Klasse im Spiel sein kann. Wenn aber die Kontrolle fest in der Klasse integriert wird beinhaltet jedes Objekt diese Kontrollstrukturen.

Szenario d

Ein Modul benutzt ein anderes Modul.

(1)



(2)

Die Schnittstellen der Programmeinheiten werden separat (unabhängig) spezifiziert und dokumentiert. Festzuhalten dass andere Module verwendet werden ist wichtig, da ein Modul nur eine Instanz hat und die Konsistenz durch eine zeitgleiche Benutzung eines anderen Modules zu Inkonsistenzen führen kann.

(3)

Jedes Modul prüft im ersten Ansatz seine eigene Konsistenz. Dass wie in (2) beschrieben Daten an anderer Stelle geändert wurden, lässt sich zwar oft, aber nicht immer feststellen.

(4)

Die Programmeinheiten sind Potentiell wiederverwendbar, wenn keine modulinternen „globalen“ Variablen verändert werden können. Sonst sollte eine zeitgleiche Benutzung nicht unbedingt stattfinden.

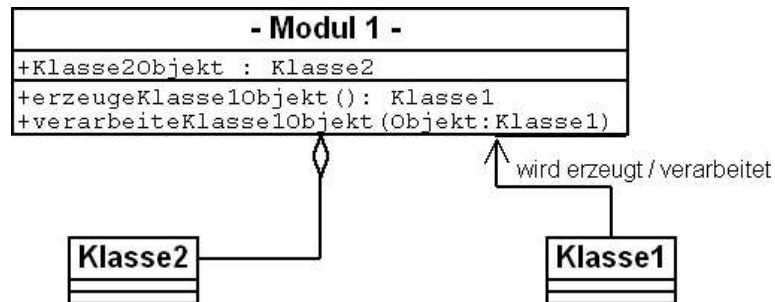
(5)

Der Ansatz ist diszipliniert und gekapselt, jedoch ist das Modul-Konzept mit Vorsicht zu genießen.

Szenario e

Ein Modul benutzt Klassen, indem es davon Objekte vereinbart, erzeugt oder benutzt.

(1)



(2)

Wie im Diagramm abgebildet wird spezifiziert welche Funktion welchen Klassentyp wie übergeben bekommt oder zurückliefert. Die Benutzung wird wie im Diagramm spezifiziert, kann jedoch in der Schnittstelle durch Einschränkung der Sichtbarkeit entfallen.

(3)

Jede Klasse sorgt in sich für seine Konsistenz und das Modul über seine eigene Konsistenz (natürlich mit berücksichtigung verwendeter Klassen).

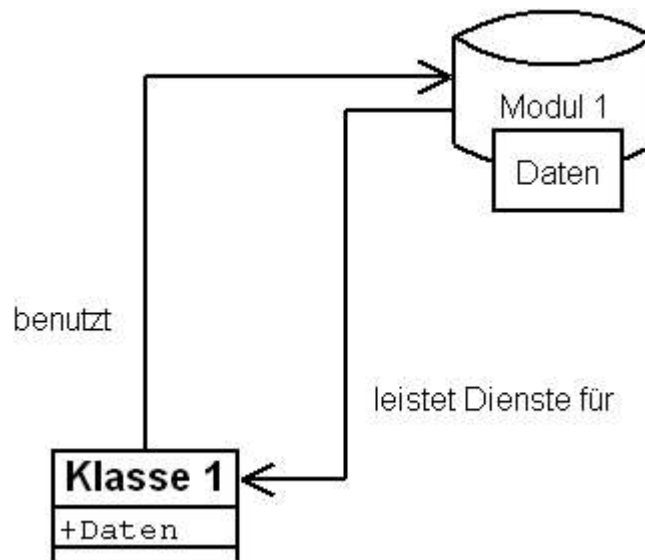
(4)

Die Klassen sind potentiell wiederverwendbar. Das Modul mit Einschränkungen auch. Siehe d.

Szenario f

Eine Klasse benutzt ein Modul, sodass ihre Objekte dieses Modul benutzen können.

(1)



(2)

In der Klasse ist zu spezifizieren/dokumentieren, dass das Modul benutzt wird.

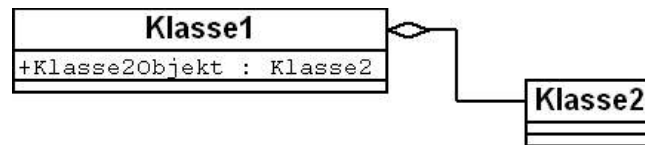
(3)-(4)

Da das Modul ähnlich globalen Daten ist, ergeben sich die gleichen Problemfälle wie in b.

Szenario g

Eine Klasse A benutzt eine Klasse B, sodass ein Objekt von A ein Objekt von B benutzen kann.

(1)



(2)

Die Klassen werden für sich spezifiziert, die Aggregation wird berücksichtigt.

(3)

Beide Programmeinheiten sind potentiell wiederverwendbar.

(4)

Es wird offensichtlich Funktionalität ausgelagert, was gut ist. Klasse2 kann durch eine Ableitung ersetzt werden, was für die Erweiterbarkeit gut ist. Klasse2 kennt Klasse1 idealerweise nicht / nur bedingt, so ist eine gute Kapselung gegeben.

Globale Daten sind der Teufel

Sie sind kaum zu kontrollieren, beliebig veränderbar und machen jedes Projekt, dass nicht 120% diszipliniert behandelt wird auf kurz oder lang inkonsistent. Kaum reproduzierbare Seiteneffekte im großen Stil sind die Folge. Es mag kurzfristig erfolgversprechend sein, eine solche Bindung verschlechtert aber die Wiederverwendbarkeit von Funktionen/Modulen/Klassen in nicht zu vernachlässigender Weise. Globale Daten zu haben ist ja nicht das Problem. Eine Klasse sollte von ihnen aber nichts wissen müssen. Sie müssen stets übergeben oder zugewiesen werden, dann bleibt die Wiederverwendbarkeit gewährleistet.

Geht man so vor, kann man auf globale Daten völlig verzichten und es würde einem nicht einmal auffallen.

3.2 Objektorientierte Programmiersprachen und Datenabstraktion

Component Pascal

Wie bereits erwähnt kommt das vorgestellte Modulkonzept bei Verwendung von „lesen- und-schreiben“-Attributen globalen Variablen gleich. Vorteil ist, da ein Modul im globalen Speicherbereich liegt, dass schnell darauf zugegriffen werden kann. Verzichtet man auf erwähnte Variablen wird das Konstrukt unflexibel aber sicherer. Dann kann man es bedingt als Funktionssammlung oder Objektlieferant benutzen.

Smalltalk / Eiffel

Das erfüllt alle Wünsche nach Kapselung. Die Klassen sind in sich geschlossen, hochgradig wiederverwendbar und können ihre Konsistenz eigenständig prüfen.

Eiffel

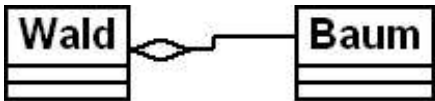
Ein prinzipieller Lesezugriff auf exportierte Attribute ist eine gute Eigenschaft, da sie für bloße Verwendung Funktionsaufrufe spart. Dass Funktionen für das Verändern solcher Attribute erstellt werden müssen sorgt dafür, dass die Klasse selbst beim setzen von Attributen ihre Konsistenz jederzeit im Griff hat.

C++, Java, C#

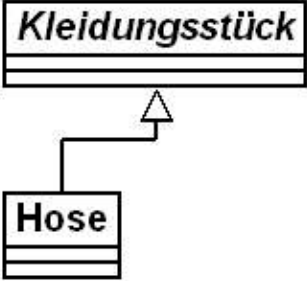
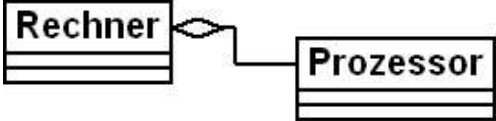
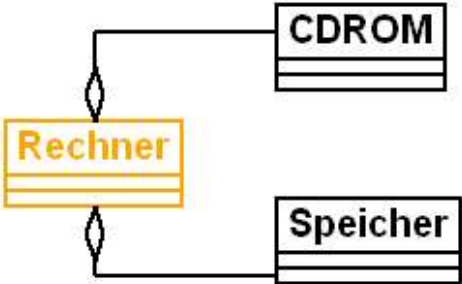
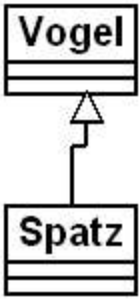
Hier hat wohl der pragmatische Ansatz triumphiert. Ich bin momentan eher unschlüssig ob diese Eigenschaft irgendwo von Bedeutung ist.

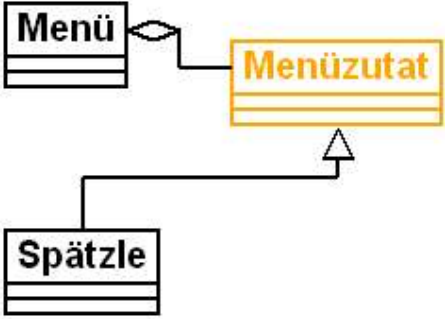
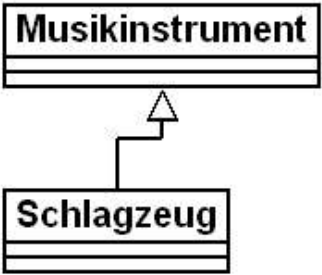
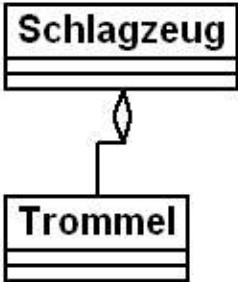
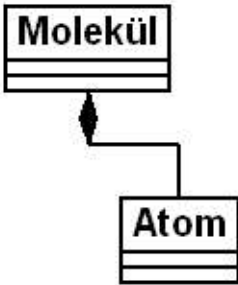
Hier tendiere ich dann zu den Programmierrichtlinien, die für solche Zwecke Methoden vorschreiben.

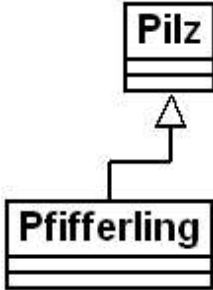
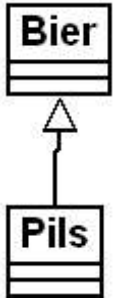
3.3 Klassenbeziehungen

<i>Dinge, Objekte, Klassen</i>	<i>UML-Diagramm</i>
Wald, Baum	 <pre> classDiagram class Wald class Baum Wald "o--" Baum </pre>

<i>Dinge, Objekte, Klassen</i>	<i>UML-Diagramm</i>
Tanne, Baum	<pre>classDiagram Tanne -- > Baum</pre> A UML class diagram showing inheritance. The class 'Baum' is at the top, and 'Tanne' is below it. A solid line with an open triangle arrowhead points from 'Tanne' to 'Baum'.
Baum, Blatt	<pre>classDiagram Baum "1" -- "*" Blatt</pre> A UML class diagram showing composition. The class 'Baum' is at the top, and 'Blatt' is below it. A solid line with an open diamond at the 'Baum' end and an arrowhead at the 'Blatt' end connects them.
Blatt, Titelblatt	<pre>classDiagram Titelblatt -- > Blatt</pre> A UML class diagram showing inheritance. The class 'Blatt' is at the top, and 'Titelblatt' is below it. A solid line with an open triangle arrowhead points from 'Titelblatt' to 'Blatt'.
Hemd, Kleidung	<pre>classDiagram Kleidung "1" -- "*" Hemd</pre> A UML class diagram showing composition. The class 'Kleidung' is at the top, and 'Hemd' is below it. A solid line with an open diamond at the 'Kleidung' end and an arrowhead at the 'Hemd' end connects them.

<i>Dinge, Objekte, Klassen</i>	<i>UML-Diagramm</i>
Hose, Kleidungsstück	 <pre> classDiagram class Kleidungsstück class Hose Kleidungsstück < -- Hose </pre>
Prozessor, Rechner	 <pre> classDiagram class Rechner class Prozessor Rechner "1" *-- "1" Prozessor </pre>
CD-ROM, Speicher	 <pre> classDiagram class Rechner class CDROM class Speicher Rechner "1" *-- "1" CDROM Rechner "1" *-- "1" Speicher </pre>
Vogel, Spatz	 <pre> classDiagram class Vogel class Spatz Vogel < -- Spatz </pre>

<i>Dinge, Objekte, Klassen</i>	<i>UML-Diagramm</i>
Spätzle, Menü	 <pre> classDiagram class Menü class Menüzutat class Spätzle Menü "1" *-- "*" Menüzutat Spätzle -- > Menüzutat </pre>
Anweisung, Zuweisung	
Zuweisung, Ausdruck	
Musikinstrument, Trommel	 <pre> classDiagram class Musikinstrument class Schlagzeug Schlagzeug -- > Musikinstrument </pre>
Trommel, Schlagzeug	 <pre> classDiagram class Schlagzeug class Trommel Schlagzeug "1" *-- "*" Trommel </pre>
Atom, Molekül	 <pre> classDiagram class Molekül class Atom Molekül "1" *-- "*" Atom </pre>

<i>Dinge, Objekte, Klassen</i>	<i>UML-Diagramm</i>
Pfifferling, Pilz	 <pre>classDiagram class Pilz class Pfifferling Pfifferling -- > Pilz</pre>
Pils, Bier	 <pre>classDiagram class Bier class Pils Pils -- > Bier</pre>

3.4 Strukturmodell: Klassifizierung oder Zerlegung

<i>Klassifizierungsstruktur</i>	<i>Bestandteilstruktur</i>
E/A-Gerät - Ausgabegerät - Drucker - Tintenstrahldrucker - Laserdrucker - Bildschirm - Eingabegerät - Scanner	Haus - Wohngeschoss - Küche - Bad - Badewanne - Dach - Dachbalken - Dachziegel

3.5 Strukturmodell: Klassifizierung und Zerlegung

<i>Klassifizierungsstruktur</i>	<i>Bestandteilstruktur</i>
Fahrzeug - Automobil - Personenkraftwagen - Lastkraftwagen - Schienenfahrzeug - Lokomotive - Straßenbahn	Fahrzeug - Tank - Motor - Getriebe - Einlassventil - Auslassventil - Motorblock - Zündkerze - Rad - Bremse
Motor - Verbrennungsmotor - Dieselmotor - Benzinmotor - Elektromotor	

3.6 Strukturmodell: Klassifizierung, Zerlegung und Assoziation

3.7 Von der Analyse zur Implementation

Klassifikation

Abbildung: Vererbung und mehrfache Vererbung

Stellt man fest, dass ein Objekt mit einem anderen so verwandt ist, dass man für beide eine gemeinsame Basis schaffen kann, wird ein Oberobjekt mit den Gemeinsamkeiten gebildet, von dem die zwei dann gemeinsame Funktionalität erben. Jedes Objekt spezialisiert diese allgemeinen Eigenschaften je nach Bedarf.

Komposition

Abbildung: Attribut, dass mehrere Objekte eines Typs sowie Methoden, die den Zugriff gewähren, beinhaltet. Kompositionen sind verbindlich, können also nicht leer sein.

Aggregation

Abbildung: Attribut, dass ein oder mehrere Objekte enthalten kann.

Aggregation ist oft unverbindlich, d.h. dass das Attribut auch leer sein kann.

Assoziation

Abbildung: Objekt wird übergeben oder zurückgegeben. Innerhalb der Klasse spielt es oft keine funktionale Rolle.

Stellt man fest, dass ein Objekt ein anderes kennen sollte, beispielsweise um Werte auszulesen und zu verarbeiten oder das assoziierte Objekt gar zu verändern, wird diese Art verwendet.

3.8 Abstrakte Klassen

„Der Mechanismus der abstrakten Klasse unterstützt die Notation eines generellen Konzepts, wie eine Form, von der nur konkretere Varianten, wie Kreis oder Quadrat, wirklich verwendet werden können.

Eine abstrakte Klasse kann auch eine Schnittstelle definieren, von der abgeleitete Klasse eine Vielzahl von Implementationen bereitstellen.

Eine abstrakte Klasse ist eine Klasse, die nur als Basisklasse einer anderen Klasse verwendet werden kann; es kann kein Objekt einer abstrakten Klasse erzeugt werden, ausser Objekte die eine Ableitung desselben sind.

Eine Klasse ist abstrakt, wenn sie mindestens eine pur virtuelle Methode besitzt.

Eine virtuelle Methode wird „pur“, indem bei der Methoden-Deklaration in der Klassen-Deklaration kein Methoden-Rumpf angegeben wird.

Eine pure virtuelle Methode muss nur definiert werden, wenn sie explizit mit einem qualifizierten Namen syntaktisch aufgerufen wird.“

Oder vereinfacht:

Eine abstrakte Klasse ist eine Roh-Vorlage für andere Klassen. Sie enthält fehlende Eigenschaften. In den davon abgeleiteten Klassen müssen diese gewissen Eigenschaften angegeben werden. Das können Methoden oder Attribute sein. Deswegen kann man von solchen abstrakten Klassen keine Objekte erzeugen; sie können ja noch nichts.

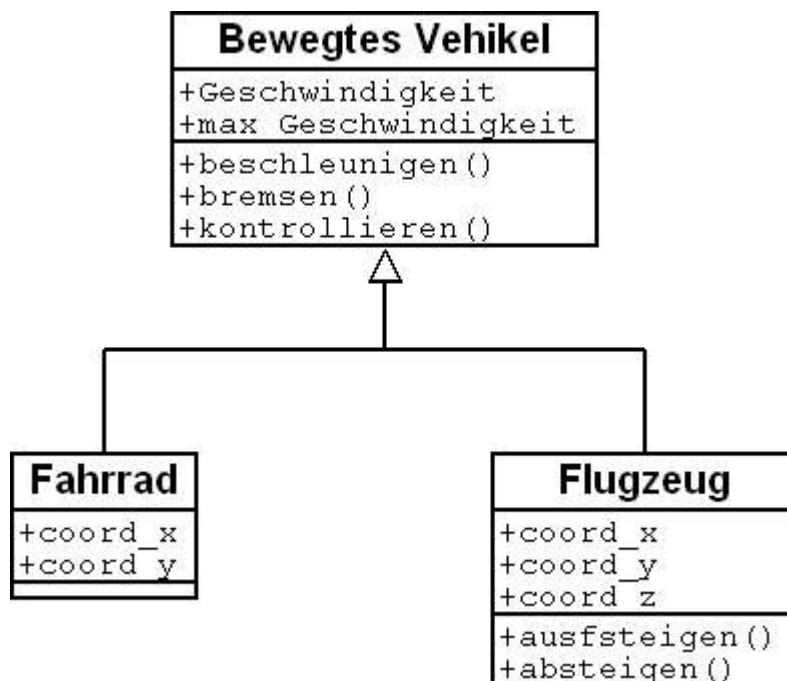
Man sagt, eine Klasse ist abstrakt, wenn mindestens eine sogenannte virtuelle Methode existiert. Das ist eine Methode, die nur aus Name und Parametern besteht und keine Implementation hat. Diese virtuelle Methode muss dann überschrieben werden, damit eine davon abgeleitete Klasse funktionieren kann.

Eine Schnittstelle (auch Interface genannt), ist eine Klasse, die nur aus virtuellen Methoden besteht. Hier sollte man aber vorsichtig sein, bevor man zu viel Angaben macht, denn der Name und die Parameter der Methoden müssen in allen abgeleiteten Klassen identisch sein, und das ist manchmal garnicht erwünscht.

3.9 Vererbung

Das vorgelegte Modell ist in sich unstimmt. Beim objektorientierten Entwurf sollen möglichst natürliche Abhängigkeiten verwendet werden. Offensichtlich haben sich die Autoren hauptsächlich an den Attributen und nicht an der Bedeutung der Objekte orientiert.

Ein Flugzeug ist **keine** erweiterte Form eines Fahrrads. Das ist grotesk.



Ich habe bewusst Elemente, die ein „Bewegtes Vehikel“ nicht zwingend haben muss in den Unterklassen erst eingeführt. Ungeachtet dessen, dass die Koordinatenangabe ein eigenes Objekt sein könnten.

So könnte man auch ein Raumschiff mit Infinite-Improbability-Drive daraus modellieren, welches gänzlich verschiedene Koordinaten benötigt.

3.10 Leitlinien für Vererbung

(1) Wiederverwenden: Bilde nie eine neue Klasse, wenn Du eine existierende Klasse wiederverwenden kannst, sei es direkt, durch Benutzen oder Erben.

Wie in 3.9 beschrieben, ist das prinzipiell richtig, man sollte jedoch auf den Kontext achten, indem das benutzte oder vererbt Objekt steht. Einer Eisenbahn Sternenkoordinaten zu geben, indem man einige davon vernachlässigt wäre sicher kein guter Ansatz.

(2) Klassifizieren: Setze Vererbung zum Generalisieren und Spezialisieren ein, d.h. um die „ist-ein“-Relation zu realisieren.

Prinzipiell ja, es darf aber nicht überhand nehmen. Vererbung in mehr als drei Ebenen ist eher schädlich. Ein Hamburger ist ein Nahrungsmittel, Ein Cheesburger ist ein Hamburger, ein Cheesburger_mit_Salat ist ein Cheesburger, ein Cheesburger_mit_Salat_und_Ketchup ist ein Cheesburger_mit_Salat... usw.

(3) Enterben: Vermeide Vererbung, die nur dem Wiederverwenden von Code dient; verwende dafür Benutzung.

Siehe (2): Ja, richtig. Siehe (1): Kontext wichtig.

(4) Herausfaktorisieren: Verbessere Vererbungsstrukturen, indem Du gemeinsame Merkmale von Unterklassen (Nachfolger-, abgeleitete Klassen) in gemeinsame Oberklassen (Vorgänger-, Basisklassen) verlagerst, ohne dabei die „ist-ein“-Relation zu verletzen.

Auch in Ordnung. Durch das bilden von sinnvollen allgemeineren Klassen, können neue Klassen einfacher implementiert werden.

(5) Abstrahieren: Lass abstrakte Klassen die gemeinsamen Abstraktionen, Rollen und Merkmale all ihrer Unterklassen erfassen.

Die Oberklasse sollte von den möglichen Implementierungen nichts wissen, da sie allgemein gehalten sein sollte.

(6) Hellsehen: Lass eine Klasse die Entwicklung zukünftiger Unterklassen nicht einschränken, solange diese die durch die Oberklassen gegebene Abstraktion konkretisieren.

Solange diese Änderungsmöglichkeit die Konsistenz der Klasse nicht betreffen kann, ok. Es gibt ja Internas, die in abgeleiteten Klassen nicht von Bedeutung sein sollten, sondern intuitiv über Methoden verwendet werden.

(7) Konzeptklassen bilden: Nutze abstrakte Klassen, um die zukünftige Entwicklung von Unterklassen geeignet zu ermöglichen und zu kontrollieren.

Ja stimmt.

(8) Schnittstellenklassen bilden: Nutze Vererbung, um polymorphes Ersetzen von Objekten der Unterklassen anstelle von Objekten der Oberklassen zu ermöglichen.

Auch richtig. Wird allerdings ein bestimmter Ableitungszweig benötigt und dies ist nur implizit angegeben, kann das zu Problemen führen.

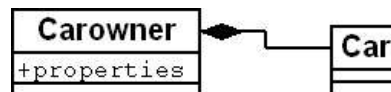
3.11 Mehrfaches Erben

Anmerkung: Das Verhalten der Mehrfachvererbung ist mit etwas erhöhtem Aufwand (selbst ohne Interfaces) mittels Aggregation und Delegation möglich.

Bild 3.2: Nicht richtig. Was ist das denn? Ein Autobesitzer ist ein Auto?

Viel mehr sollte der Autobesitzer ein oder mehrere Autos besitzen.

Ganz simpel:



Oder etwas abgedrehter, aber möglicherweise unzureichend:

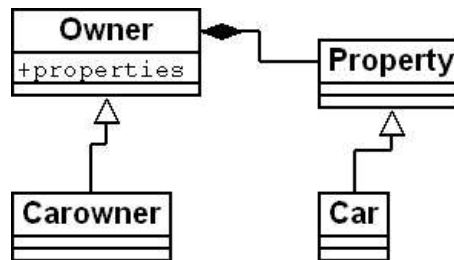
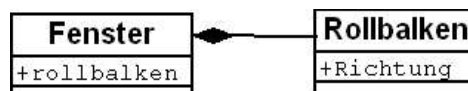


Bild 3.3: Nicht richtig. Bei mehrfacher Vererbung werden auch gleiche Attribute mehrfach eingebettet. Das ist ganz böse. Zumindest in C++ muss man selbige dann sogar konkret ansprechen.

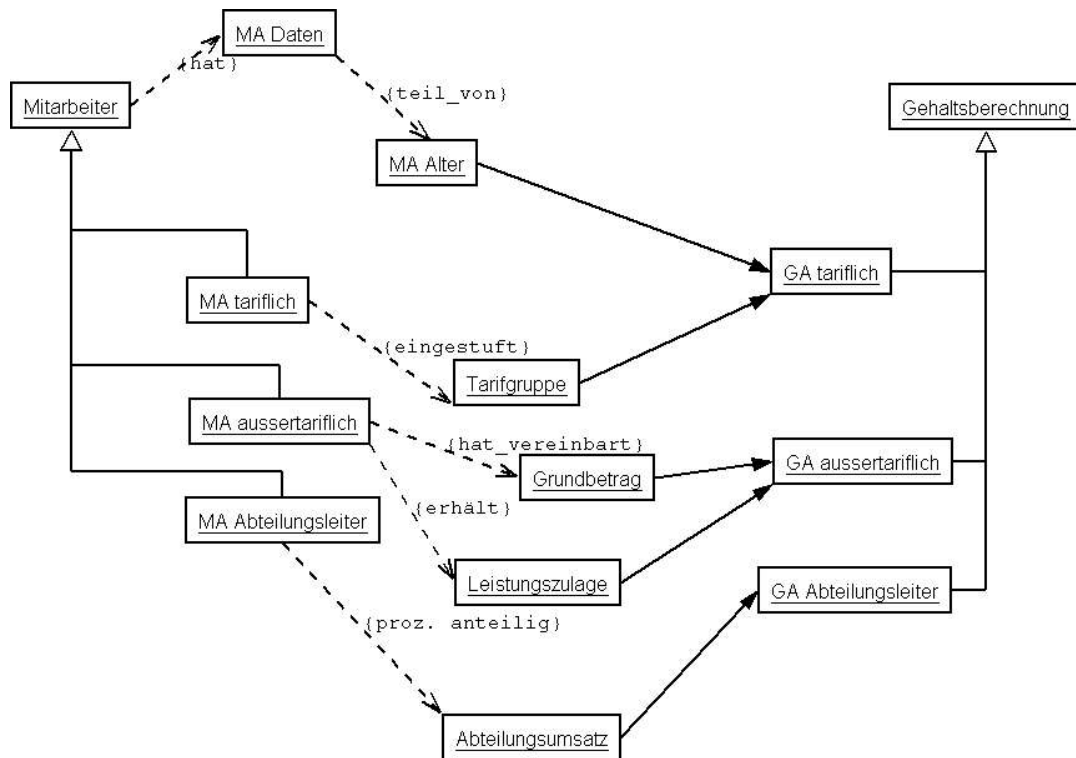


Mit Fabriken oder Erbauern können so vorgefertigte Objekte erzeugt werden.

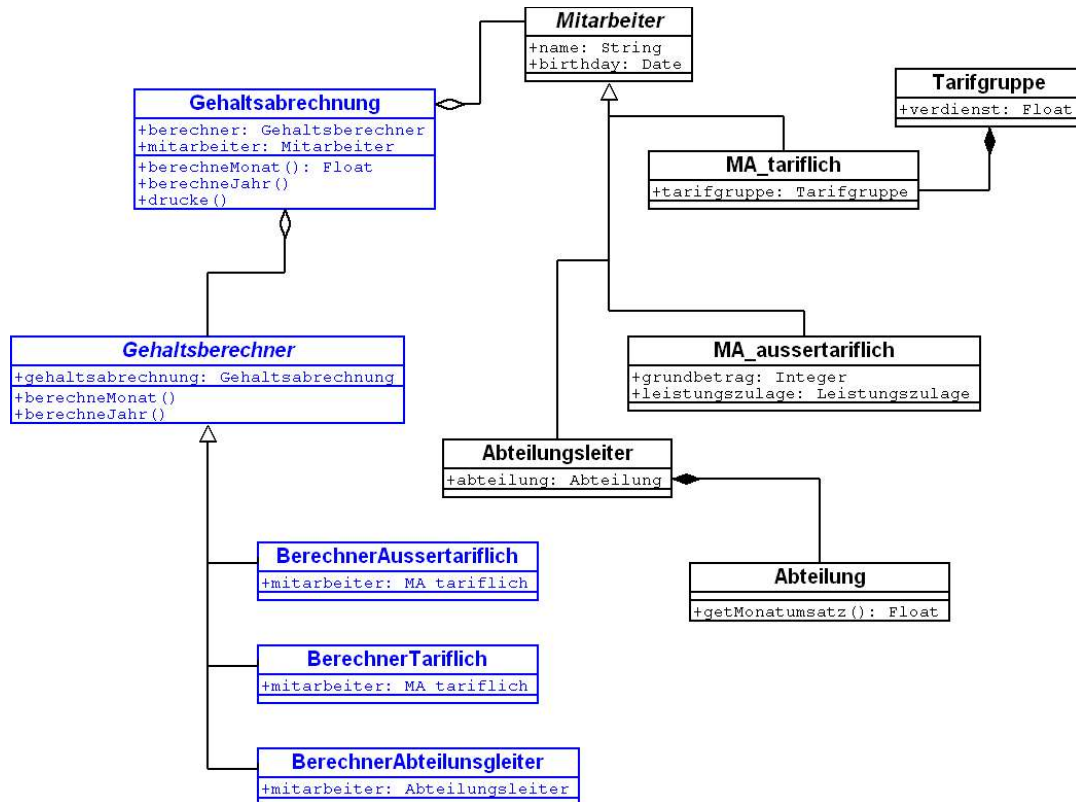
Fabrik::erzeugeFensterMitHorizontalemRollbalken();

Aufgabe 3.12 Entwurf von Klassen - Gehaltsbuchhaltung

Grafisches Strukturmodell



Klassendefinitionen in UML



Erläuterungen

Abstrakte Klasse

Die Mitarbeiter-Klasse ist eine abstrakte Klasse. Sie muss erst spezialisiert werden. Im Beispiel durch „tariflicher Mitarbeiter“, „ausertariflicher Mitarbeiter“ und „Abteilungsleiter“. Warum muss sie spezialisiert werden? Bei jeder sorte Mitarbeiter berechnet sich das Gehalt (teils statisch, teils dynamisch) aus anderen Gegebenheiten. Die Spezialisierung soll diese Gegebenheiten beinhalten.

Polymorphie

Die Gehaltsabrechnung beinhaltet je ein von der Mitarbeiter-Klasse abgeleitetes Objekt. Mittels eines polymorphen Gehaltsberechners (je nach Gehaltsgruppe) werden über die Berechne-Funktion andere Gegebenheiten in der Berechnung berücksichtigt.

Würde der Mitarbeiter gewisse Teile seines Gehalts selbst berechnen, könnten die Berechner-Unterklassen entfallen, und es gäbe wirkliche Polymorphie, denn dann würde es egal sein, was für eine Art Mitarbeiter es ist.

Dynamisches Binden

Das Beispiel bietet hierfür nicht viel Nährboden, da konkret mit Spezialisierungen gearbeitet wird. Würde ein Mitarbeiter sein eigenes Gehaltsabrechnungsobjekt erzeugen und der Berechner es aufnehmen, so wäre nicht bekannt, welche Art Unter-Klasse er nun erhalten hat. Die Schnittstelle bleibt jedoch die selbe, und so kann es benutzt werden.