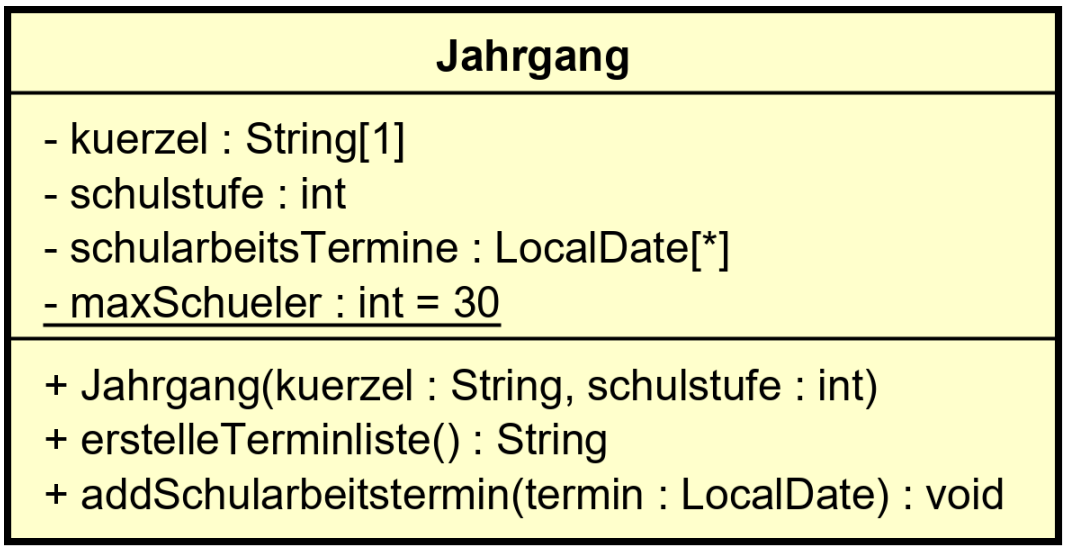
Vom Klassendiagramm zum Java-Code

Attribute

Betrachten wir folgendes Beispiel. Es soll eine Java-Klasse erstellt werden, welche allgemeine Attribute zu einem Jahrgang einer Schule speichert. Es soll das Kürzel des Jahrgangs gespeichert werden können (z.B. „3AHIT“), die Schulstufe (z.B. 3), die maximale Anzahl an Schülern pro Klasse (welche für alle Jahrgänge gilt) sowie mehrere vereinbarte Schularbeitstermine. Auf Basis dieser Anforderung wurde folgendes Klassendiagramm erstellt:



UML Klassendiagramm: Beispiel Jahrgang

Für alle Sichtbarkeiten wurde `private` gewählt. Die Angabe eines Kürzels ist verpflichtend, d.h. die Angabe eines Werts ist *nicht* optional. Daher wurde die Multiplizität „[1]“ für `kuerzel` gewählt. Die Schulstufe ist eine normale Zahl. Da es mehrere Schularbeitstermine gibt, wurde die Multiplizität „[*]“ gewählt, d.h. theoretisch wäre es möglich, dass es auch gar keine Schularbeitstermine gibt. In Java kann man zum Abspeichern von Datumsangaben die Klasse `LocalDate` verwenden (Nachfolger von `Date`). Nachdem die maximale Schüleranzahl für alle Jahrgänge gelten soll, wurde sie als statisches Attribut (Klassenattribut) modelliert. Standardmäßig ist dieser Wert `30`, dieser kann sich jedoch auch während der Programmausführung ändern.

Neben einem Konstruktor wurde eine Methode `erstelleTerminliste` definiert, welche alle Schularbeitstermine als String zurückliefert. Mit `addSchularbeitstermin` können neue

Schularbeitstermine dem Attribut `schularbeitsTermine` hinzugefügt werden.

Eine passende Implementierung in Java sieht folgendermaßen aus:

Java-Code: Beispiel Jahrgang

```
import java.time.LocalDate;
import java.util.Arrays;

public class Jahrgang {
	private String kuerzel;
	private int schulstufe;
	private LocalDate[] schularbeitsTermine;
	private static int maxSchueler = 30;

	public Jahrgang(String kuerzel, int schulstufe) {
		if (kuerzel == null) {
			throw new IllegalArgumentException(
			"Fuer kuerzel muss ein Wert angegeben werden!");
		}
		this.kuerzel = kuerzel;
		this.schulstufe = schulstufe;
		this.schularbeitsTermine = new LocalDate[0];
	}

	public String erstelleTerminliste() {
		StringBuilder sb = new StringBuilder();
		for (int i = 0; i < schularbeitsTermine.length; i++) {
			sb.append(schularbeitsTermine[i].toString());
			sb.append("\n");
		}
		return sb.toString();
	}

	public void addSchularbeitstermin(LocalDate termin) {
		schularbeitsTermine = Arrays.copyOf(schularbeitsTermine,
					schularbeitsTermine.length + 1);
		schularbeitsTermine[schularbeitsTermine.length - 1] =
					termin;
	}
}
```

Multiplizität [1] in Java

Nachdem für `kuerzel` ein Wert angegeben werden muss (Multiplizität **[1]**), wird im Konstruktor überprüft, ob **`null`** übergeben wird, und ggf. eine `IllegalArgumentException` über den Befehl `throw` geworfen. Somit wird verhindert, dass `kuerzel` keinen Wert speichert: Versucht ein aufrufendes Programm, `null` zu übergeben, wird der Programmfluss durch die Exception unterbrochen. Dies müsste auch in entsprechenden `set`-Methoden überprüft werden. Bei primitiven Datentypen entfällt eine solche Überprüfung, da diese ja nicht `null` sein können.

Multiplizität [*] in Java

Für `schularbeitsTermine` sollen mehrere Werte (Multiplizität **[*]**) gespeichert werden können. Dies kann über **Arrays** gelöst werden.

Demnächst lernen wir eine weitere Methode kennen, die sogar noch besser dafür geeignet ist: Collections. Für dieses Beispiel reicht uns jedoch vorerst ein Array. Wäre eine Mindestgröße gefordert (z.B. [1..*]), so könnte man beispielsweise im Konstruktor bereits Werte als zusätzlichen Parameter zur Erzeugung verlangen.

Arrays.copyOf

Das Array wächst in der Methode `addSchularbeitstermin` dynamisch um jeweils ein Element pro Aufruf. Dabei werden die alten Werte des Arrays in ein neues Array kopiert, das um 1 Wert größer ist. Hierfür wird die statische Methode `copyOf` der Hilfsklasse `Arrays` verwendet. Der erste Parameter ist das Ursprungs-Array und der zweite Parameter gibt die neue Größe an. Zurückgeliefert wird eine Kopie des Arrays mit der neuen Größe. Sieht man sich `copyOf` genauer an, merkt man, dass es sich hier um keine „normale“ Methode handelt. Sie kann nämlich für unterschiedliche Datentypen verwendet werden, indem sie mit einem Datentyp parametrisiert wird. Details dazu lernen wir bald unter dem Stichwort „Generics“ bzw. „Generizität“ kennen.

Klassenattribute sind in Java static

`maxSchueler` soll für alle `Jahrgang`-Objekte gelten, d.h. es soll nur 1 gemeinsamer Wert gespeichert werden und *nicht* 1 Wert pro Objekt. Daher wurde dieses Attribut mit dem **`static`** Schlüsselwort versehen. Ist also im UML-Klassendiagramm ein Attribut **unterstrichen**, handelt es sich dabei in Java immer um ein `static`-Attribut.

Startwert als Initialisierung in Java

Außerdem soll `maxSchueler` mit dem Startwert `30` initialisiert werden. Dies kann leicht erreicht werden, indem es gleich bei der Definition mit dem Wert `30` **initialisiert** wird.

Folgende Main-Methode testet das Verhalten von Jahrgang:

```
public static void main(String[] args) {
	Jahrgang j = new Jahrgang("3AHIT", 3);
	j.addSchularbeitstermin(LocalDate.of(2019, 11, 11));
	j.addSchularbeitstermin(LocalDate.of(2019, 12, 23));
	j.addSchularbeitstermin(LocalDate.of(2020, 1, 13));
	System.out.println(j.erstelleTerminliste());
}
```

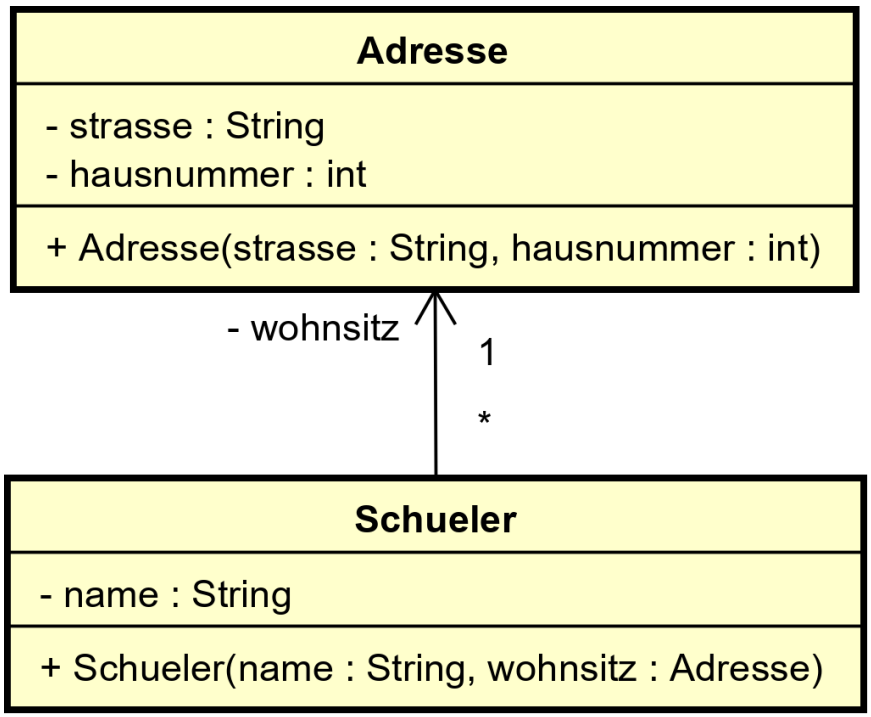
Die Ausgabe lautet in diesem Fall:

```
2019-11-11
2019-12-23
2020-01-13
```

Assoziationen

Eine Assoziation ist eine Verbindung zwischen zwei Klassen. Das bedeutet in Java im Allgemeinen, dass eine Klasse ein Attribut vom Typ der anderen Klasse speichert. Betrachten wir folgendes Beispiel:

UML Klassendiagramm: Assoziation



In diesem Beispiel gibt es zwei Klassen: `Adresse` und `Schueler`. Die Assoziation ist so zu lesen: Ein Schüler hat genau einen Wohnsitz. Eine Adresse kann beliebig vielen Schülern zugeordnet sein (0 bis unendlich). Es könnten also beispielsweise auch zwei Schüler im selben Haus wohnen.

Die Pfeilrichtung bedeutet, dass der `Schueler` seine `Adresse` kennt. Hat man also ein `Schueler`-Objekt, kann man leicht seine/ihre `Adresse` ermitteln. Umgekehrt ist dies nicht der Fall: Hat man eine `Adresse`, kann man nicht automatisch auch alle zugehörigen `Schueler` -Objekte ermitteln.

Eine Umsetzung in Java sieht folgendermaßen aus:

Java-Code: Assoziation

```
public class Adresse {
	private String strasse;
	private int hausnummer;

	public Adresse(String strasse, int hausnummer) {
		this.strasse = strasse;
		this.hausnummer = hausnummer;
	}
}
```

```java
public class Schueler {
	private String name;
	private Adresse wohnsitz;

	public Schueler(String name, Adresse wohnsitz) {
		if (wohnsitz == null) {
			throw new IllegalArgumentException("Fuer wohnsitz
					muss ein Wert angegeben werden!");
		}
		this.name = name;
		this.wohnsitz = wohnsitz;
	}
}
```

Pfeilrichtung in Java

Nachdem jeder `Schueler` genau einen Wohnsitz hat und der `Schueler` seinen Wohnsitz kennen soll, löst man dies über ein normales Attribut. Da die Multiplizität [1] ist, muss sichergestellt werden, dass auch wirklich ein `Adresse`-Objekt übergeben wurde. Durch dieses Attribut ist nun sichergestellt, dass der Pfeilrichtung im Klassendiagramm gefolgt werden kann: Hat man ein `Schueler`-Objekt, kann man über dieses auf das `Adresse`-Objekt zugreifen (z.B. mittels einer Setter- oder Getter-Methode).

Wenn ein Schüler mehrere Wohnsitze haben können soll, würde man dies über ein Array vom Typ `Adresse[]` lösen (wie bei bei `schularbeitsTermine` von `Jahrgang`).

Assoziationen in Java abbilden

Bei Assoziationen gibt es daher drei Dinge, auf die man achten muss:

- **Pfeilrichtung**: Grundsätzlich gilt, dass man der Pfeilrichtung auch im Code folgen können muss. Das bedeutet, dass die Klasse, auf die mit dem Pfeil gezeigt wird (in diesem Beispiel `Adresse`), als Attribut in der Ursprungsklasse gespeichert werden muss.
- **Multiplizitäten**: Als nächstes muss man sich die Multiplizität auf der Seite der Pfeilspitze ansehen. Wenn es [1] oder [0..1] ist, reicht ein normales Attribut, das beim Speichern ggf. auf `null` überprüft werden muss. Sollen mehrere Werte gespeichert werden ([*], [1..*]), verwendet man Arrays oder Collections (siehe später).
- **Rollenname**: Idealerweise verwendet man im Klassendiagramm Rollen- oder Assoziationsnamen. Diese eignen sich meist hervorragend für die Benennung des jeweiligen Attributs.

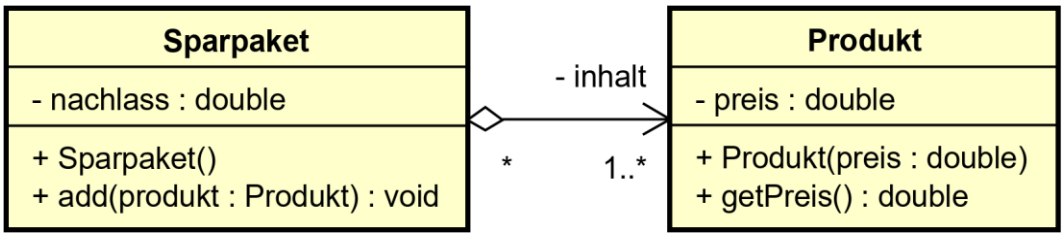
Bidirektionale Assoziationen

Handelt es sich um bidirektionale Assoziationen (zwei Pfeilspitzen), überlegt man sich die obigen Punkte für beide Seiten. Die Herausforderung dabei ist, dass beide Richtungen bzw. beide Attribute synchron gehalten werden müssen, was sehr aufwändig ist. Daher vermeidet man normalerweise bidirektionale Assoziationen.

Aggregation

Eine Aggregation ist eine spezielle Assoziation, die eine Teil-Ganzes-Beziehung abbildet. Man kann also „A ***besteht aus*** B“ sagen. Beispielsweise besteht ein Sparpaket aus mehrere Produkten:

UML Klassendiagramm: Aggregation



In diesem Beispiel können beliebig viele `Produkte`, aber zumindest eines (Multiplizität [1..*] rechts) zusammen in einem `Sparpaket` verkauft werden. Außerdem kann ein `Produkt` auch in mehreren `Sparpaketen` angeboten werden (Multiplizität [*] links). `Produkte` können aber auch einzeln verkauft werden, d.h. sie können also auch ohne `Sparpaket` existieren ([*] inkludiert die 0). Man kann aber trotzdem jedenfalls sagen, dass ein `Sparpaket` aus mehreren `Produkten` *besteht*, deswegen ist es eine Aggregation.

Pfeil von Aggregationen

Diese „lockere“ Besteht-Aus-Abhängigkeit ist charakteristisch für eine Aggregation. Dies zeigt sich auch dadurch, dass in der `add` Methode ein `Produkt`-Objekt übergeben wird, d.h. das Aggregat ist nicht verantwortlich für die Erzeugung seiner Teile. Bei der Aggregation zeigt der Pfeil üblicherweise vom Ganzen zum Teil. Der Diamant befindet sich immer auf der Seite des Ganzen.

Folgender Java-Code stellt eine korrekte Implementierung dar:

Java-Code: Aggregation

```
public class Produkt {
	private double preis;

	public Produkt(double preis) {
		this.preis = preis;
	}

	public double getPreis() {
		return preis;
	}
}
```

```java
import java.util.Arrays;

public class Sparpaket {
	private double nachlass;
	private Produkt[] inhalt;

	public Sparpaket(double nachlass, Produkt p) {
		if (p == null) {
			throw new IllegalArgumentException("Fuer p muss ein
					Wert angegeben werden!");
		}
		this.nachlass = nachlass;
		inhalt = new Produkt[1];
		inhalt[0] = p;
	}

	public void add(Produkt produkt) {
		inhalt = Arrays.copyOf(inhalt, inhalt.length + 1);
		inhalt[inhalt.length - 1] = produkt;
	}
}
```

Aggregation in Java

Auch eine Aggregation wird typischerweise über ein **Array** (noch besser: Collection) gelöst. Nachdem der Pfeil von `Sparpaket` zu `Produkt` führt, muss das `Sparpaket` seine `Produkt`e kennen. Daher wird in der Klasse `Sparpaket` ein Array vom Typ `Produkt` als Attribut definiert. So wird sichergestellt, dass dem Pfeil von `Sparpaket` zu `Produkt` gefolgt werden kann. Nachdem `Produkt` die Rolle `inhalt` hat, wird das Attribut auch gleich so benannt. Die Sichtbarkeit ist wie im Klassendiagramm `private`.

Multiplizität [1..*] in Java

Nachdem die Multiplizität auf der `Produkt`-Seite **[1..*]** ist, muss sichergestellt werden, dass immer zumindest ein gültiges `Produkt` im Array ist. Dies kann hier gelöst werden, indem gleich im Konstruktor überprüft wird, ob das erste übergebene `Produkt` **nicht `null`** ist. Erlaubt eine Klasse auch das Löschen oder Bearbeiten von Objekten, muss hier ebenfalls verhindert werden, dass das letzte gültige Objekt verloren geht.

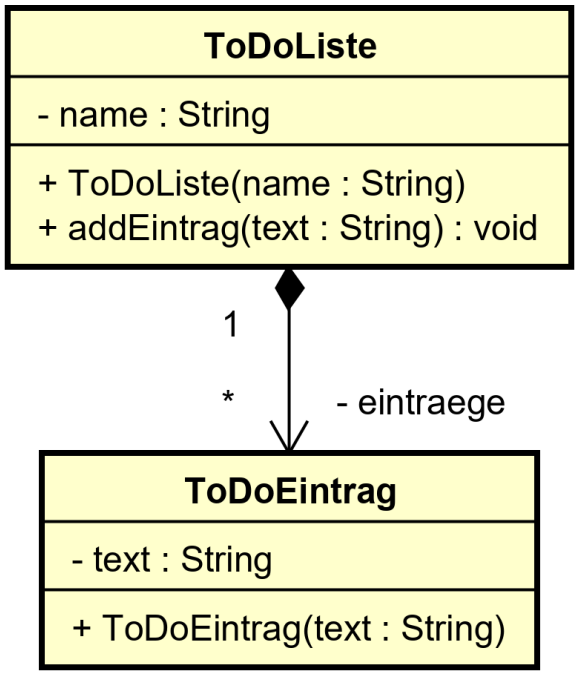
Unterschied zwischen Assoziation und Aggregation

Zusammengefasst ist der Hauptunterschied zwischen einer normalen [1]-[*] oder [*]-[*] Assoziation und einer Aggregation die **semantische** Bedeutung: Wenn man sagen kann „A ***besteht aus*** B“, handelt es sich höchstwahrscheinlich um eine Aggregation. Die Implementierung in Java unterscheidet sich dabei nicht.

Komposition

Die Komposition ist eine strengere Form der Aggregation, was auch im Code berücksichtigt werden muss. Betrachten wir folgendes Beispiel einer einfachen ToDo-Liste:

UML Klassendiagramm: Komposition



Komposition in Java

Eine `ToDoListe` *besteht aus* mehreren `ToDoEintrag`-Objekten. Anders als bei der Aggregation kann das Teil jedoch nicht ohne einem Ganzen existieren. Jeder Eintrag braucht daher eine `ToDoListe`, weshalb die Multiplizität auf der Seite des Ganzen **immer [1]** sein muss. Wird die ganze `ToDoListe` gelöscht, so werden auch die Einträge gelöscht (außer sie werden vorher in eine andere `ToDoListe` geschoben). Ein Eintrag kann außerdem nur einer einzigen `ToDoListe` zugeordnet sein.

Verwaltung der Teile

Man sieht außerdem, dass die `addEintrag`-Methode **kein fertiges `ToDoEintrag`-Objekt** übernimmt, sondern nur die notwendigen **Informationen zur Erzeugung** dessen (in diesem Fall der `text`). Das Ganze ist bei einer Komposition nämlich immer für die Verwaltung (Erzeugung, Löschung, …) seiner Teile zuständig.

Eine Implementierung in Java sieht folgendermaßen aus:

Java-Code: Komposition

```
public class ToDoEintrag {
	private String text;

	public ToDoEintrag(String text) {
		this.text = text;
	}

	public String getText() {
		return text;
	}

	public void setText(String text) {
		this.text = text;
	}
}

import java.util.Arrays;

public class ToDoListe {

	private String name;
	private ToDoEintrag[] eintraege;

	public ToDoListe(String name) {
		this.name = name;
		this.eintraege = new ToDoEintrag[0];
	}

	public void addEintrag(String text) {
		eintraege = Arrays.copyOf(eintraege, eintraege.length + 1);
		eintraege[eintraege.length - 1] = new ToDoEintrag(text);
	}

	public String toString() {
		StringBuilder sb = new StringBuilder();
		sb.append(this.name).append("\n");
		for (int i = 0; i < eintraege.length; ++i) {
			sb.append(eintraege[i].getText()).append("\n");
		}
		return sb.toString();
	}

	public static void main(String[] args) {
		ToDoListe t = new ToDoListe("Hallo Welt");
		t.addEintrag("Einkaufen");
		t.addEintrag("Hausuebung machen");
		System.out.println(t.toString());
	}
}
```

Die Komposition wird ähnlich wie die Aggregation realisiert. Es wird wieder ein **Array** zur Verwaltung der Einträge verwendet. In diesem Fall ist die Multiplizität auf der Eintrag-Seite **[*]**, d.h. man muss nicht sicherstellen, dass zumindest ein `ToDoEintrag` im Array vorhanden ist.

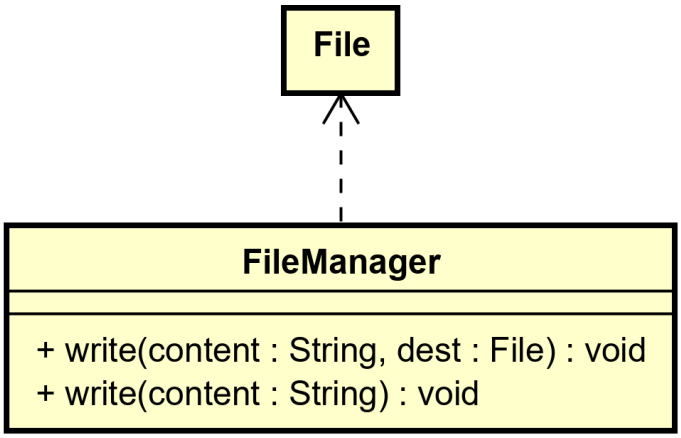
Erzeugung von Teilen im Kompositum

Der wesentliche Unterschied zur normalen Aggregation und Assoziation findet sich in diesem Beispiel in der `addEintrag`-Methode. Diese übernimmt nämlich nur den `text` als `String` und erzeugt selbstständig den Eintrag über den `new`-Befehl. In diesem simplen Beispiel macht es kaum Unterschied. Bei komplexeren Themenstellungen könnten aber beim Erzeugen zusätzliche Operationen für die Verwaltung nötig sein.

Abhängigkeiten

Man stößt öfters auf Pfeile, die wie eine Assoziation aussehen, aber gestrichelt dargestellt werden:

UML Klassendiagramm: Abhängigkeiten



Eine Abhängigkeit ist **keine Assoziation** und bedeutet nur, dass die Klasse von einer anderen Klasse in irgendeiner Form *abhängt*. Dies kann beispielsweise sein:

- Die Klasse wird als Parameter verwendet
- Es wird ein Objekt dieser Klasse in einer Methode erzeugt (aber nicht in einem Attribut gespeichert)
- Eine statische Methode dieser Klasse wird aufgerufen
- Eine Exception vom Typ dieser Klasse wird gefangen

Das Gerüst eines passenden Java-Beispiels sieht aus wie folgt:

Java-Code: Abhängigkeiten

```java
import java.io.File;

public class FileManager {
	public void write(String content, File dest) {
		// Write to File dest...
	}

	public void write(String content) {
		File f = new File("default.txt");
		// Write to default File f...
	}
}
```

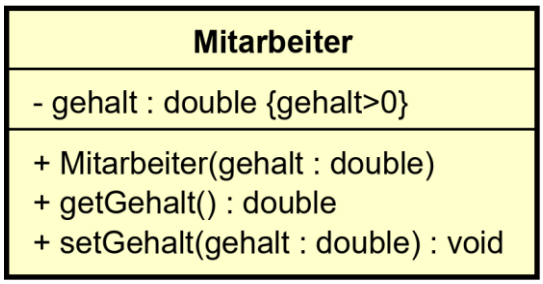
`FileManager` hat in diesem Beispiel zwei Abhängigkeiten von `File`. Einmal wird `File` als Paramter verwendet und einmal wird ein Objekt von `File` erzeugt. Da es aber **kein Attribut** vom Typ `File` gibt, handelt es sich um **keine Assoziation**.

Ein gestrichelter Pfeil bedeutet also nur, dass man von dieser Klasse in gewisser Weise abhängt. Wird die Klasse geändert, muss die abhängige Klasse ebenfalls ggf. angepasst werden. Abhängigkeiten zeichnet man nur ein, wenn diese wirklich wichtig ist, da das Diagramm sonst schnell unübersichtlich wird.

Erweiterte Kompetenzen

Zusicherungen

Nachdem in Java bis auf den Datentyp keine weiteren Einschränkungen direkt bei der Definition des Attributs angegeben werden können, müssen Zusicherungen überall dort geprüft werden, wo das Attribut verändert wird. Dies betrifft meistens die **Setter** und **Konstruktoren**. Folgendes Diagramm zeigt die Klasse Mitarbeiter:



UML Klassendiagramm: Zusicherungen

Eine Java-Implementierung sieht folgendermaßen aus:

Java-Code: Zusicherungen

```
public class Mitarbeiter {

	private double gehalt;

	public Mitarbeiter(double gehalt) {
		this.setGehalt(gehalt);
		this.gehalt = gehalt;
	}

	public double getGehalt() {
		return gehalt;
	}

	public void setGehalt(double gehalt) {
		if (gehalt <= 0) {
			throw new IllegalArgumentException("Das Gehalt muss
                                        groesser als 0 sein!");
		}
		this.gehalt = gehalt;
	}
}
```

Im Setter (und daher auch im Konstruktor) wird überprüft, ob die Einschränkung eingehalten wird und ggf. eine Exception geworfen, wenn das `gehalt` kleiner oder gleich 0 würde. Solche Überprüfungen müssten je nach Zusicherung an allen Stellen geschehen, an denen die Zusicherung verletzt werden könnte.

Zusicherungen für Assoziationen

Zusicherungen, die für Assoziationen gelten, werden genauso wie bei Attributen überprüft. Es muss daher immer dann geprüft werden, wenn sich die Objektbeziehungen (z.B. neues Produkt zur Liste hinzugefügt) oder die beteiligten Attribute ändern (z.B. neuer Preis für Produkt oder Sparpaket).

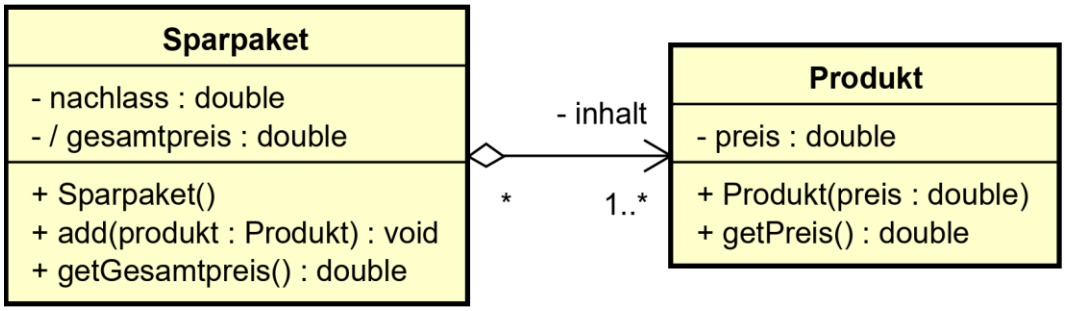
Abgeleitete Attribute

Getter mit oder ohne Attribut

Bei abgeleiteten Attributen gibt es mehrere Möglichkeiten, wie diese in Java realisiert werden können. Zunächst muss man sich die Frage stellen, ob überhaupt ein **eigenes Attribut** dafür nötig ist oder ob ein einfacher **Getter** reicht, der den Wert **jedes Mal neu berechnet**. Die Lösung über ein Attribut ist bei komplexen Berechnungen beim Lesen performanter, muss aber bei Änderungen immer neu berechnet werden. Als Faustregel kann man daher sagen: Wird das Attribut oft gelesen, kann man durchaus ein eigenes Attribut verwenden. Ändert sich der Wert hingegen häufig, ist ein Getter ohne Attribut besser.

Betrachten wir nochmals unser Beispiel mit dem Sparpaket:

UML Klassendiagramm: Abgeleitete Attribute



Der `gesamtpreis` lässt sich jederzeit anhand der `preis`-Attributwerte von `Produkt` und `nachlass` berechnen. Eine Lösung als eigenes Attribut könnte folgendermaßen aussehen:

Java-Code: Abgeleitete Attribute

```
import java.util.Arrays;

public class Sparpaket {
	private double nachlass;
	private double gesamtpreis;
	private Produkt[] inhalt;

	public Sparpaket(double nachlass, Produkt p) {
		if (p == null) {
			throw new IllegalArgumentException("Fuer p muss ein
					Wert angegeben werden!");
		}
		this.nachlass = nachlass;
		inhalt = new Produkt[1];
		inhalt[0] = p;
		gesamtpreis = p.getPreis() * nachlass;
	}

	public void add(Produkt produkt) {
		inhalt = Arrays.copyOf(inhalt, inhalt.length + 1);
		inhalt[inhalt.length - 1] = produkt;
		gesamtpreis += produkt.getPreis() * nachlass;
	}

	public double getGesamtpreis() {
		return gesamtpreis;
	}
}
```

In diesem Fall wird das `gesamtpreis`-Attribut immer dann aktualisiert, wenn ein neues `Produkt` der Liste hinzugefügt wird. Falls Einträge auch gelöscht oder bearbeitet werden können, müsste man hier ebenfalls jedes Mal den `gesamtpreis` aktualisieren.

Alternativ könnte der Wert für `gesamtpreis` auch jedes Mal neu berechnet werden – dann müsste man kein eigenständiges Attribut speichern und aktuell halten:

```
	public double getGesamtpreis() {
		double gesamtpreis = 0;
		for (int i = 0; i < inhalt.length; i++) {
			gesamtpreis += inhalt[i].getPreis();
		}
		return gesamtpreis * nachlass;
	}
```

Habe ich es verstanden?

- Ich kann Attribute aus UML-Diagrammen inklusive Datentypen, Sichtbarkeiten, Multiplizitäten, Startwerten und `static` als Java-Code umsetzen
- Ich kann Assoziationen mit unterschiedlichen Multiplizitäten, Navigierbarkeiten und Rollennamen als Java-Code realisieren
- Ich kann Aggregationen mit unterschiedlichen Multiplizitäten, Navigierbarkeiten und Rollennamen als Java-Code umsetzen
- Ich kann Kompositionen mit unterschiedlichen Multiplizitäten, Navigierbarkeiten und Rollennamen in Java realisieren
- Ich kann Abhängigkeiten in Java-Code identifizieren und erklären
- (*Erweitert*) Ich kann Zusicherungen für Attribute und Assoziationen in Java umsetzen
- (*Erweitert*) Ich kann abgeleitete Attribute in Java umsetzen