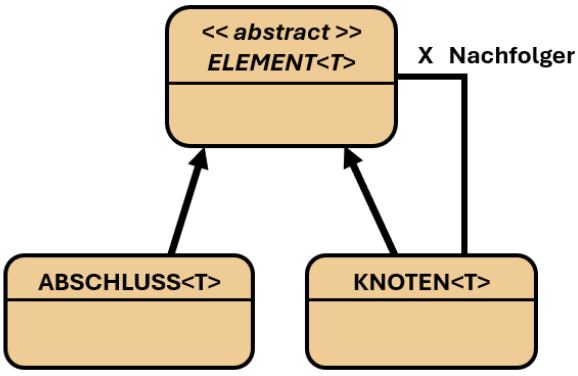
Das Entwurfsmuster Kompositum (Composite-Pattern)

Problem: Ohne dieses Entwurfsmuster gibt es sehr viele Fallunterscheidungen, die prüfen, ob man „schon am Ende angekommen“ ist oder ob sich schon etwas in der Struktur befindet oder nicht.

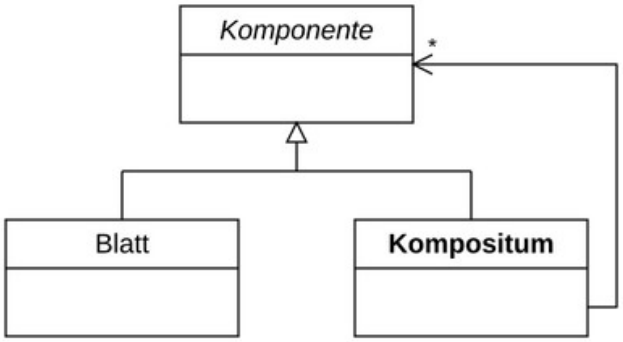
Idee: ***Ein Teil des Ganzen sieht genauso aus, wie das Ganze selbst***.

Lösung: Einführen der Oberklasse ***ELEMENT*** und einer weiteren Klasse ***ABSCHLUSS*** Solche Elemente hängt man z. B. als Nachfolger ein, wenn es eigentlich keinen Nachfolger gibt.

Konkret



Allgemein



Beispiele für die Anwendung des Kompositums: Dateisysteme (Ordner und Dateien) oder die grafische Benutzeroberfläche von Java (Alle Elemente wie Schaltflächen und Textfelder sind Spezialisierungen der Klasse Component) → siehe Kapitel zur Softwareentwicklung

Einschub: Abstrakte Klassen und Methoden

Klassen und Methoden können in Java den Modifikator ***abstract*** bekommen. Bei Klassen bedeutet das abstract, dass man ***keine Objekte von dieser Klasse erzeugen kann***. Dies kann bei Oberklassen Sinn ergeben, wenn man z. B. nicht möchte, dass ein allgemeines undefiniertes Tier erzeugt werden soll. Die Attribute und Methoden werden weiterhin normal vererbt.

In abstrakten Klassen können ***abstrakte Methoden definiert*** werden. Diese werden genauso wie die Methode beim Interface angegeben ***(ohne Methodenrumpf)***. Abstrakte Methoden einer Oberklassen ***müssen von den Unterklassen implementiert werden*** (→ wie beim Interface)

Unterschiede zwischen abstrakten Klassen und Interfaces

	Abstrakte Oberklasse	**Interface**
Attribute	Kann Attribute haben und vererben	Hat keine Attribute
Methoden	Kann normale Methoden mit Methodenrumpf bzw. Methodeninhalt haben, die vererbt werden. Kann auch abstrakte Methoden haben, die von den Unterklassen implementiert werden müssen.	Hat nur Methodensignaturen und Klassen mit diesem Interface müssen diese Methoden implementieren.
Logik	Abstrakte Oberklassen definieren zu gegebenen Klassen eine allgemeinere Spezies, welche aber nicht existiert. (→ Generalisierung)	beschreiben also ein oder mehrere Methoden, die nötig sind, um eine Fähigkeit „nachzurüsten“.

Umbau der rekursiven Datenstruktur unter Verwendung des Entwurfsmusters Kompositum

- **Allgemein**: In der Oberklasse ***ELEMENT*** sind **alle** Methoden als abstrakte Methodensignaturen aufgelistet. Damit verfügt der ***ABSCHLUSS*** genau über dieselben Methodensignaturen wie der ***KNOTEN***, nur dass die Methodenrümpfe in jeder der Klassen unterschiedlich implementiert werden.
- **Referenzattribute**: Der Abschluss besitzt keine Attribute. Nur der Knoten verfügt über ***einen Inhalt*** und über ***einen Nachfolger***. Dieser ist vom Typ ***ELEMENT***. Hat ein KNOTEN eigentlich keinen Nachfolger, so hängt man ihm einen ABSCHLUSS-Objekt als Nachfolger an.
  Somit hat deine LISTE ein ***erstes Element vom Typ ELEMENT***. Ist die Datenstruktur leer, so ist das erste Element ein ABSCHLUSS.
- **Methoden**: Der ABSCHLUSS ***stellt dennoch alle Methoden***, die er eigentlich nicht braucht (z.B. nenneNachfolger()) zur Verfügung. Der Rumpf dieser Methoden wird so gestaltet, dass dadurch in anderen Klassen Fallunterscheidungen vermieden werden können.

Aufgabe 1: Klassendiagramm

Zeichne ein Klassendiagramm zu einer LISTE mit Kompositum.

Aufgabe 2: Umsetzung Kompositum (allgemeiner Teil und Attribute)

Dupliziere dein Projekt ohne Git-Verknüpfung: Erstelle ein neues BlueJ-Projekt und kopiere alle .java-Dateien.

Benenne dein Projekt um in: ***Datenstruktur mit Kompositum***

Erstelle die Klassen ELEMENT und ABSCHLUSS und baue dein Projekt bezüglich **Vererbung, Attribute/Datentypen und Konstruktoren** – wie oben beschrieben – um!

(Hinweis: Kopieren des Projektordners und anschließendes Löschen der Datei team.defs und des versteckten .git-Ordners ist nur am eigenen Gerät möglich)

Aufgabe 3: Umsetzung Kompositum (einfache Methoden)

Gehe die folgenden Methoden in der Klasse ABSCHLUSS, KNOTEN und LISTE durch und prüfe, ob diese angepasst werden müssen:

inhalt(), setzeNachfolger(), vorneEntfernen(), nachfolger(), vorneEinfügen()

Aufgabe 4: Umsetzung Kompositum (Methoden der LISTE)

Gehe die folgenden Methoden in der Klasse ABSCHLUSS, KNOTEN und LISTE durch und prüfe, ob diese angepasst werden müssen:

gibLänge(), suchenUndIndexZurückgeben(…)

Aufgabe 5: `hintenEinfügen(T t) : void, hintenEntfernen() : T`

a) Zeichne ein Objektdiagramm einer Liste mit 2 Inhalten „A“ und „B“ unter Nutzung des Entwurfsmusters Kompositum.

b) Ermittle das Problem, wenn man hinten etwas einfügen möchte? (Tipp: Wann weiß man, dass man hinten angekommen ist und wo muss man einfügen?)

Grundtrick zum Einfügen an der Stelle vor dem Objekt: (in allgemeiner Form)

Ich gebe mein Anliegen rekursiv an meinen Nachfolger weiter und speichere mir seine Antwort. Ich wiederum gebe meinem Vorgänger die Antwort auf sein Anliegen.

c) Implementiere die Methode **`hintenEinfügen(T t) : void`** neu.

d) Zeichne ein Sequenzdiagramm zu der LISTE aus Teilaufgabe 5a), wenn die Methode **`hintenEinfügen(new KUNDE("C"))`** aufgerufen wird.

Aufgabe 6:

Implementiere anhand desselben Grundtrick die folgenden Methoden **neu**:

a) **`suchenUndEntfernen(String info) : void`** (ohne Zurückgeben des Objekts)

b) **`entfernenAn(int i) : void`** (ohne Zurückgeben des Objekts)

c) **`einfügenAn(T t, int i) : void`**

d) **`einfügenVorElement(T t) : void`**

e) **`einfügenSortiert(T t) : void`**

Aufgabe 7:

Implementiere zum Thema LISTE Abituraufgaben!

Mebis-Link zum Prüfungsarchiv aller Informatikabituraufgaben

Schülerlösungsvorschläge zu den Abituraufgaben vom Rupprecht-Gymnasium