Datenstruktur LISTE

Eine ***LISTE*** ist eine SCHLANGE mit weiteren Funktionen wie z.B.

- **Einfügen an bestimmter Stelle**
- **Sortiertes Einfügen**
- **Suchen nach einem bestimmten Element**
- **Entfernen eines bestimmten Elements**

Da eine LISTE ***eine Erweiterung einer SCHLANGE*** ist, werden wir sie als Unterklasse von SCHLANGE implementieren.

Bevor wir dies allerdings tun, werden wir uns ein neues Design erarbeiten, das uns viele lästige Fallunterscheidungen (z.B. ob ein Nachfolger existiert) ersparen wird …

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***. (oft bei rekursiven Datenstrukturen)

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

- Der ***ABSCHLUSS*** verfügt genau über dieselben Methodensignaturen wie der ***KNOTEN***, nur dass die Methodenrümpfe in jeder der Klassen unterschiedlich implementiert werden. In der Oberklasse ***ELEMENT*** sind all diese Methoden als abstrakte Methodensignaturen aufgelistet.
- Deine SCHLANGE / STACK / LISTE hat ein ***erstes Element vom Typ ELEMENT***. Ist die Datenstruktur leer, so ist das erste Element ein ABSCHLUSS.
- Nur der KNOTEN verfügt über einen Nachfolger. Dieser ist vom Typ ELEMENT. Hat ein KNOTEN eigentlich keinen Nachfolger, so hängt man ihm einen ABSCHLUSS als Nachfolger an.
  Nur der KNOTEN hat einen Inhalt.
- 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.

Ein erstes Beispiel:

Ein KNOTEN gibt bei laenge() die Antwort seines Nachfolgers, erhöht um Eins, zurück. Ein ABSCHLUSS gibt nun z.B. den sinnvollen Wert 0 zurück und erspart dem KNOTEN die Fallunterscheidung, ob er einen Nachfolger hat. Aber auch die SCHLANGE selbst braucht keine Fallunterscheidung mehr bei der Methode laenge(). Sie gibt die Frage an ihr erstes Element weiter. Ist die SCHLANGE leer, so ist ihr erstes Element ein ABSCHLUSS und der antwortet mit dem Wert 0.

In vielen weiteren Situationen kann man durch geschickte Gestaltung der Methodenrümpfe in der Klasse ABSCHLUSS in den anderen Klassen den Code von Fallunterscheidungen befreien.

Ein zweites Beispiel:

Dadurch, dass die LISTE (SCHLANGE, STACK) nun immer ein erstes Element enthält, auch wenn sie leer ist (dann ist das erste Element ein ABSCHLUSS), kann es keine NullpointerException mehr geben, wenn man bei einer leeren LISTE (SCHLANGE, STACK) auf das erste Element zugreift. Diese Fallunterscheidung ist also auch überflüssig, solange der ABSCHLUSS eine entsprechend sinnvolle Antwort auf z.B. nenneInhalt() liefert, nämlich null.

Ein drittes Beispiel:

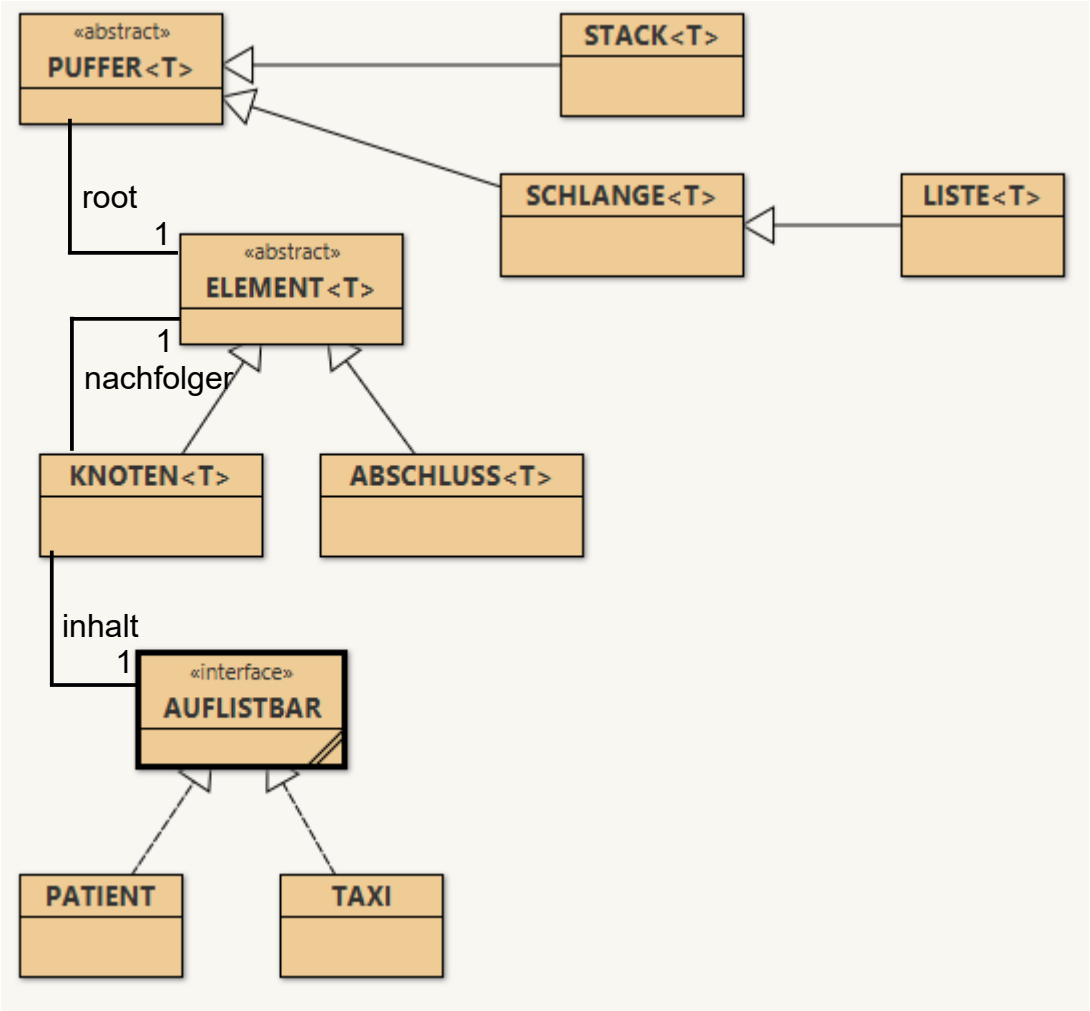
Eigentlich braucht der ABSCHLUSS keine Methode nenneNachfolger(). Wegen der Vererbungshierarchie muss der ABSCHLUSS aber über diese Methode verfügen. Gäbe ein ABSCHLUSS einfach null zurück, so hätte man die Methode zwar implementiert, aber keinen Vorteil dadurch erlangt. Deshalb gibt ein ABSCHLUSS als Nachfolger sich selbst zurück (oder einen neuen Abschluss). Dadurch kann man bei einer leeren LISTE (SCHLANGE, STACK) nicht nur mit dem Anfang, sondern auch mit dessen Nachfolger (und jedem weiteren Nachfolger) rechnen. Hierdurch wird in der Klasse LISTE (SCHLANGE, STACK) in einigen Methoden eine weitere Fallunterscheidung überflüssig.

Aufgabe 1:

a) *Sieh dir noch einmal das Klassendiagramm deiner bisherigen Implementierung an. Was musst du alles ändern? Wo wird die bisherige Klasse KNOTEN benötigt? Was kann alles bleiben, wie es ist? Wo sind Fallunterscheidungen, die vermutlich abgeschafft werden können durch die neue Struktur?*

b) ***Beginne nun ein neues Projekt und implementiere alles noch einmal von vorne, diesmal allerdings unter Berücksichtigung des Composite-Pattern.*** *Konzentriere dich zunächst nur auf PUFFER, STACK und SCHLANGE. Die LISTE nimmst du dir erst vor, wenn STACK und SCHLANGE nachweisbar funktionieren.*

Auf der nächsten Seite siehst du das Klassendiagramm und die Klassenkarten. Um alle hier neu hinzukommenden Methoden kümmerst du dich erst bei der LISTE!



PUFFER<T>

- # root : ELEMENT
- + PUFFER()
- + istLeer() : boolean
- + laenge() : int
- + pop() : T
- + *push(T) : void*

<< Interface >>

AUFLISTBAR

- + *info() : String*
- + *istGleich(interface AUFLISTBAR) : boolean*
- + *istGroesserAls(interface AUFLISTBAR) : boolean*
- + *istKleinerAls(interface AUFLISTBAR) : boolean*

ELEMENT<T>

- + ELEMENT()
- + *indexVon(info : String, i : int) : int*
- + *inhalt() : T*
- + *ipop(int) : T*
- + *ipush(T, int) : boolean*
- + *laenge() : int*
- + *nachfolger() : ELEMENT*
- + *push(T) : KNOTEN*
- + *setzeNachfolger(ELEMENT) : void*
- + *sortiertEinfuegen(T) : KNOTEN*

KNOTEN<T>

- - inhalt : T
- - nachfolger : ELEMENT
- + KNOTEN(T)
- + indexVon(info : String, i : int) : int
- + infoGeben() : String
- + inhalt() : T
- + ipop(int) : T
- + ipush(T, int) : boolean
- + laenge() : int
- + nachfolger() : ELEMENT
- + push(T) : KNOTEN
- + setzeNachfolger(nachfolger : ELEMENT) : void
- + sortiertEinfuegen(T) : KNOTEN

LISTE<T>

- + LISTE()
- + indexVon(info : String) : int
- + indexVonIter(info : String) : int
- + init() : LISTE<TAXI>
- + ipop(int) : T
- + ipopIter(int) : T
- + ipush(T, int) : boolean
- + ipushIter(T, int) : boolean
- + sortiertEinfuegen(T) : void

ABSCHLUSS<T>

- + ABSCHLUSS()
- + indexVon(info : String, i : int) : int
- + inhalt() : T
- + ipop(int) : T
- + ipush(T, int) : boolean
- + laenge() : int
- + nachfolger() : ELEMENT
- + push(T) : KNOTEN
- + setzeNachfolger(e : ELEMENT) : void
- + sortiertEinfuegen(T) : KNOTEN

SCHLANGE<T>

- + SCHLANGE()
- + push(T) : void
- + pushIter(T) : void

STACK<T>

- + STACK()
- + push(T) : void

Aufgabe 2: **ipop(int i) ipopIter(int i)**

Diese Methode soll ein bestimmtes Element aus der LISTE entfernen. Der Parameter i gibt dabei dessen Position an. Bei unserer Variante soll die 1 das erste Element der Liste (sprich das root-Element sein)

Die Methode gibt den Inhalt des i-ten KNOTENS als generischen Datentyp T zurück oder null, falls es keinen i-ten KNOTEN gibt.

a) *Da bei dieser Aufgabe vorher bekannt ist, wie weit „gezählt" werden muss, bietet sich hier eine iterative Vorgehensweise mit Hilfe einer Zählschleife an. Implemetiere ipopIter(int i).*

b) *Überlege wie eine rekursive Variante der ipop(int i) mit dem Composite-Pattern (ohne null-Überprüfungen) aussehen würde.* ***Modelliere*** *und programmiere.*

Aufgabe 3: **ipushIter(T inhalt, int i):boolean ipush(T inhalt, int i):boolean**

Diese Methode soll ein Element an einer bestimmten Position in der Liste einfügen. Der Parameter i gibt dabei dessen Position an. Bei unserer Variante soll die 0 „vor" dem root-Element entsprechen, die 1 nach dem root-Element usw.

Die Methode gibt true zurück, falls es möglich war, an der Position i in die LISTE einzufügen, ansonsten false.

a) *Auch hier bietet sich eine iterative Vorgehensweise mit Hilfe einer Zählschleife an. Implemetiere ipushIter(T t, int i).*

b) *Überlege wie eine rekursive Variante der ipush(T t, int i) mit dem Composite-Pattern (ohne null-Überprüfungen) aussehen würde.* ***Modelliere*** *und programmiere.*

Aufgabe 4: **indexVon(String info):int indexVonIter(String info):int**

Diese Methode soll prüfen, ob ein Element mit dieser Information (z.B. Patienten-Name, Taxi-Autonummer, …) im Puffer ist. Vergleich von Strings: Siehe Java-API-Doc String.compareTo(…)

Zurück gegeben wird der Index dieses Elements im Puffer (Zählung konsistent mit Aufgabe 2 und 3, d.h. das root-Element ist 1 usw.) oder -1, falls es ein solches Element nicht gibt.

a) *Überlege wie eine iterative Variante der indexVonIter(String info) : int funktionieren könnte. Programmiere indexVonIter(String info) : int.*

b) *Überlege dir, wie man bei der rekursiven Variante den Index zählen kann. Implemetiere indexVon(String info) : int. Tipp: Hilfsparameter*

Aufgabe 5: **sortiertEinfuegen(T t) : void**

Diese Methode soll ein Element vom Typ T sortiert in die LISTE einfügen. Hierzu geht man natürlich davon aus, dass auch die eventuell bereits in der LISTE vorhandenen Elemente sortiert (aufsteigend) eingefügt wurden!

***Bevor wir diese Methode in der Klasse LISTE implementieren** können, brauchen wir die Gewissheit, dass alle Elemente, die in eine LISTE eingefügt werden können, über Vergleichs-Methoden verfügen:*

istGleich(...) , istGroesserAls(...) , istKleinerAls(...)

*Dies kann man dadurch realisieren, dass wir im **Interface AUFLISTBAR** solche Methoden zusätzlich zur Methode info() fordern. Als Übergabeparameter wählt man hier das Interface AUFLISTBAR, da der generische Typ T hier nicht bekannt ist.*

*Zur konkreten Realisierung des Vergleichs in den **Klassen, welche AUFLISTBAR implementieren**, zieht man die „info“ (String) eines Elements heran und vergleicht sie mit Hilfe der Methode compareToIgnoreCase(...) bzw. compareTo(...) der Klasse String (siehe JAVA API online – Klasse String) mit der „Info“ des anderen Elements.*

*Anschließend kann in der **Klasse LISTE** die Methode sortiertEinfuegen(T t) implementieren.*
Hier bietet es sich allerdings an, dass die Klasse LISTE ihr erstes Element neu setzt und zwar auf das, was sie von ihrem ersten Element als Antwort erhält.

*In der **Klasse ABSCHLUSS** wird aus dem einzufügenden Element ein neuer KNOTEN gebaut und dieser zurückgegeben. So erhält man eine sinnvolle Reaktion einer leeren LISTE. Aber auch eine sinnvolle Reaktion eines ABSCHLUSSes am Ende einer nicht leeren LISTE ist damit sichergestellt. (s. nächster Absatz)*

*In der **Klasse KNOTEN** wird zunächst geprüft, ob das neue Element kleiner (Tipp: String API) ist als das Element dieses KNOTENs. Ist das der Fall, so wird ein neuer KNOTEN mit dem neuen Element erzeugt und zurückgegeben. In allen anderen Fällen wird das Problem zunächst an den Nachfolger weiter gereicht (setze den Nachfolger neu auf das, was der Nachfolger antwortet) und anschließend eine Referenz auf sich selbst (dieser Knoten) an den Vorgänger (bzw. die LISTE) zurück gegeben (ich bleibe dein Nachfolger bzw. dein erstes Element).*