1.6 Das Entwurfsmuster KOMPOSITUM

In der LISTE stellen wir bei vielen Methoden fest, dass wir immer wieder überprüfen, ob die LISTE leer ist oder ein Nachfolger existiert.
Wir betrachten auszugsweise Methoden der allgemeinen LISTE:

VorneEinfuegen(DATENELEMENT d) in LISTE

anfang == null	
Wahr	Falsch
anfang = new KNOTEN(d)	KNOTEN k = new KNOTEN(d) k.NachfolgerSetzen(anfang) anfang = k

VorneEntfernen() in LISTE

anfang == null	
Wahr	Falsch
return null	DATENELEMENT t = anfang anfang = anfang.NachfolgerGeben() return t

HintenEinfuegen(DATENELEMENT d) in LISTE

anfang == null	
Wahr	Falsch
KNOTEN k = new KNOTEN(d) anfang = k	anfang.HintenEinfuegen(d)

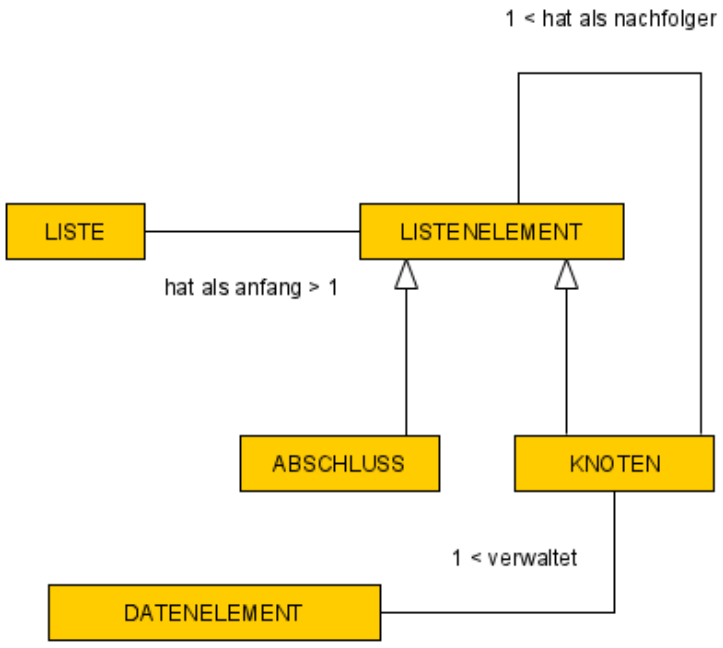
HintenEinfuegen(DATENELEMENT d) in KNOTEN

nachfolger == null	
Wahr	Falsch
KNOTEN k = new KNOTEN(d) NachfolgerSetzen(k)	nachfolger.HintenEinfuegen(d)

Dieser immer wiederkehrende Mechanismus kann dadurch vermieden werden, dass wir ein Element in die LISTE einfügen, das als leerer ABSCHLUSS-Knoten fungiert. Der ABSCHLUSS sagt uns dabei, wo das Ende der LISTE liegt. Dadurch fällt das Abfragen ob der Nachfolger oder Anfang der LISTE leer ist weg, da dort immer das ABSCHLUSS Element wartet, falls dies der Fall ist.
Es stellt sich nun die Frage, wie wir diese Idee in der Praxis am besten umsetzen. Dabei stellt man fest, dass auf zwei wichtige Punkte ankommt:

1. Der „anfang“ der LISTE hat einen festen Datentyp, d.h. beide Elemente ABSCHLUSS und KNOTEN müssen mit diesem Datentyp ansprechbar sein
2. Beide Klassen ABSCHLUSS und KNOTEN sollten idealerweise das Vorhandensein gewisser Methoden garantieren

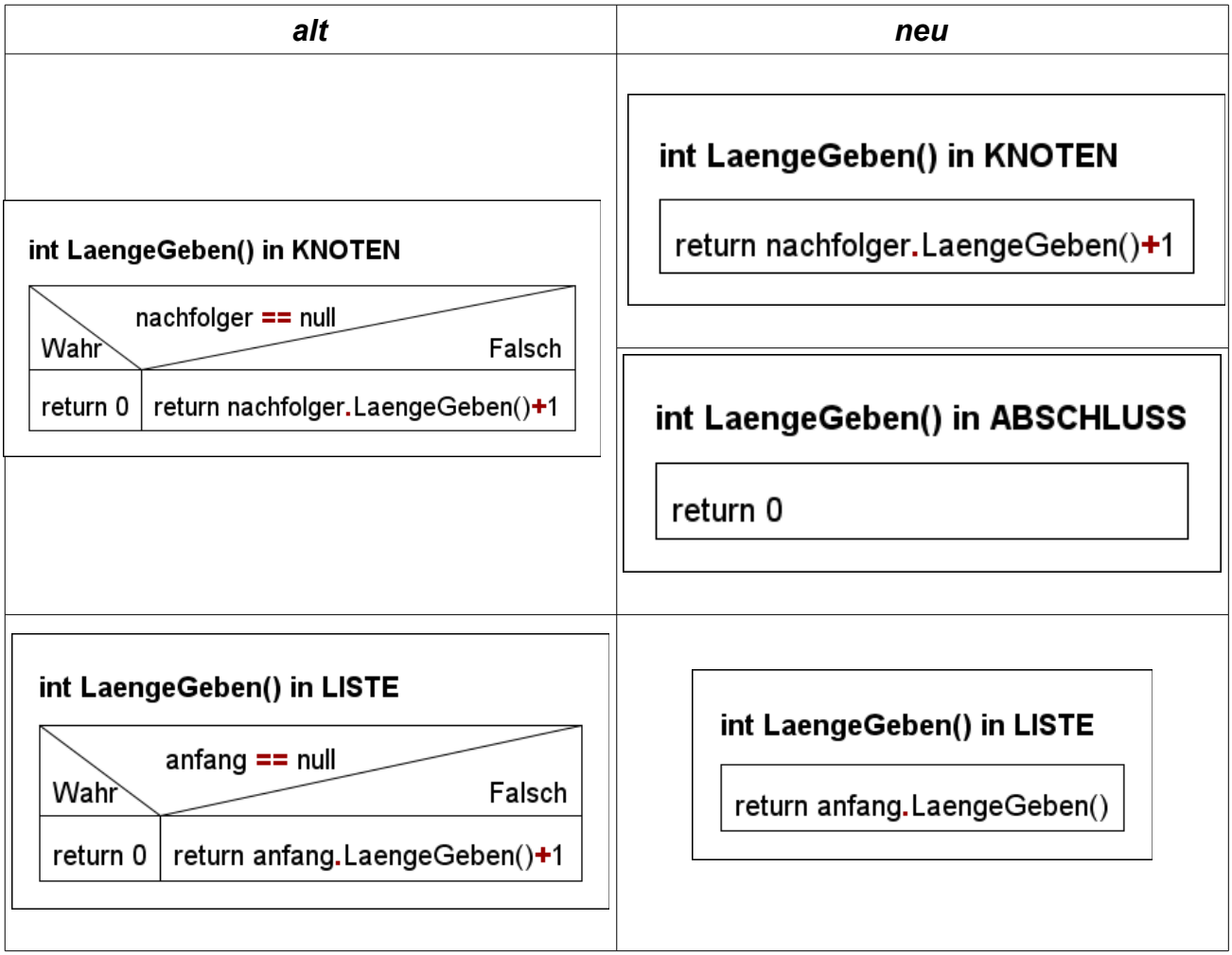
Aus diesen beiden Punkten folgt, dass wir am besten die Vererbung nutzen. Es ergibt sich nebenstehendes Klassendiagramm für das sogenannte KOMPOSITUM:



Die Methoden Umsetzung

In der Umsetzung der Methoden können wir meist die Struktogramme von früher einfach auf die beiden Klassen ABSCHLUSS und KNOTEN verteilen.

z.B. int LaengeGeben()



Es fällt auf, dass durch das Vorhandensein eines ABSCHLUSS in jeder LISTE, der Code sich im Detail leicht verändert. Der Code wechselt hier in der LISTE von
`return anfang.LaengeGeben()+1`
zu
`return anfang.LaengeGeben()`
da sonst immer die Länge 1 ausgegeben würde, obwohl danach vielleicht nur der ABSCHLUSS und somit eine leere LISTE kommen würde.

Manche andere Methoden sind schwerer umzustellen. Wenn wir bisher KNOTEN hinten eingefügt haben, haben wir rechtzeitig die Verbindung zum Vorgänger erhalten. Nun laufen wir bis zum ABSCHLUSS und haben ein Problem, da wir keine Verbindung zum Vorgänger besitzen. Um trotzdem die Verbindungen zu erhalten, brauchen wir in der HintenEinfuegen Methode nun einen Rückgabetyp. Damit gibt immer der nachfolger an seinen Vorgänger die Information zurück, wer jetzt sein Nachfolger ist. Dies ändert sich in der Mitte der LISTE nicht, jedoch beim ABSCHLUSS. Wir sehen das in den entstehenden Struktogrammen:

alte Version

HintenEinfuegen(DATENELEMENT d) in LISTE

anfang == null	
Wahr	Falsch
KNOTEN k = new KNOTEN(d) anfang = k	anfang.HintenEinfuegen(d)

HintenEinfuegen(DATENELEMENT d) in KNOTEN

nachfolger == null	
Wahr	Falsch
KNOTEN k = new KNOTEN(d) NachfolgerSetzen(k)	nachfolger.HintenEinfuegen(d)

neue Version

LISTENELEMENT HintenEinfuegen(DATENELEMENT d) in LISTE

```
anfang = anfang.HintenEinfuegen(d)
```

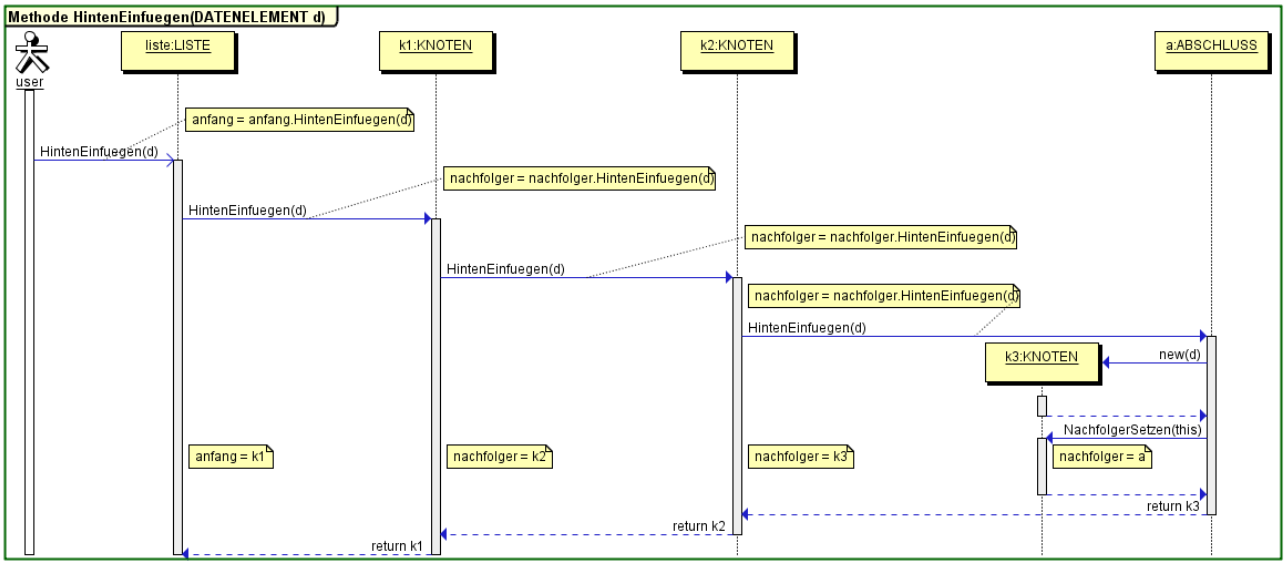
LISTENELEMENT HintenEinfuegen(DATENELEMENT d) in KNOTEN

```
nachfolger = nachfolger.HintenEinfuegen(d);
return this; //gibt sich selbst weiter!
```

LISTENELEMENT HintenEinfuegen(DATENELEMENT d) in ABSCHLUSS

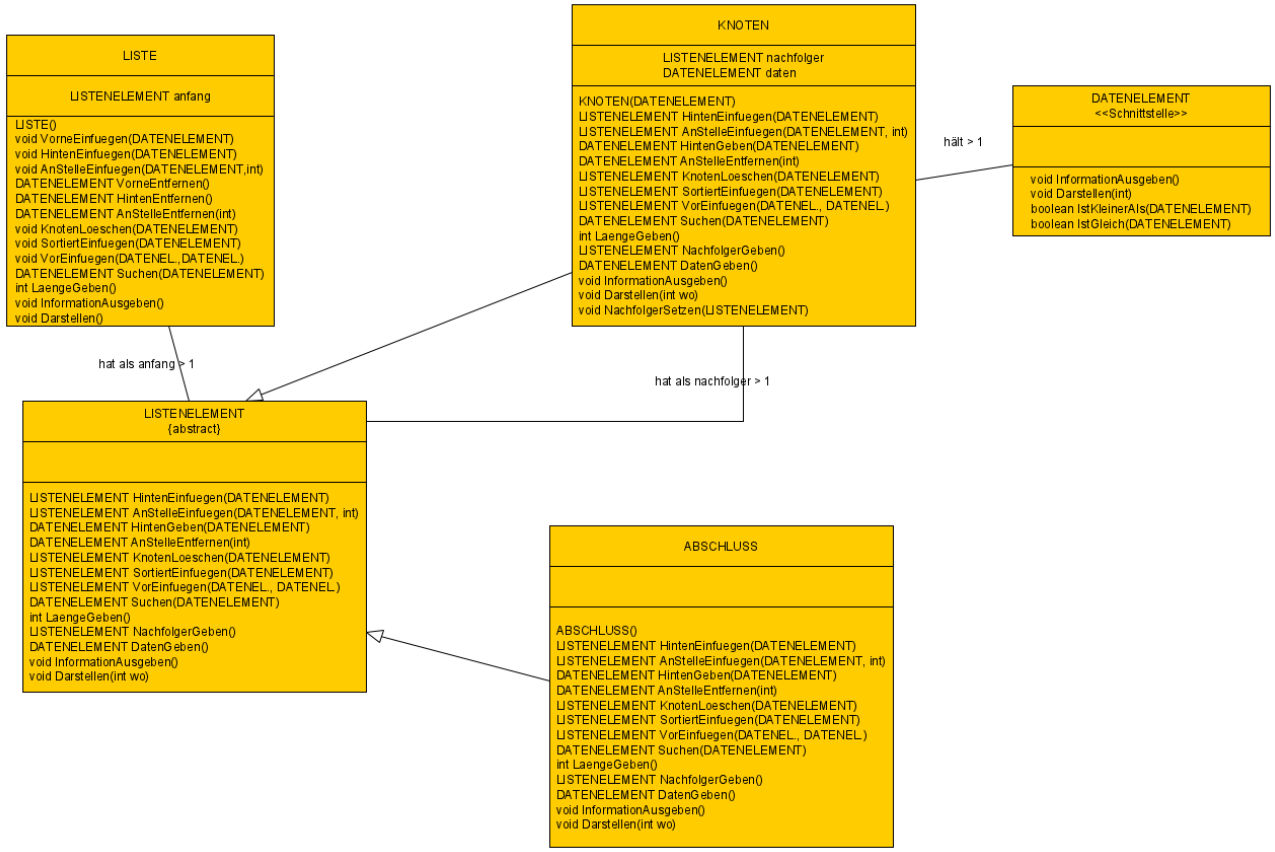
```
KNOTEN k = new KNOTEN(d);
k. NachfolgerSetzen(this);
return k;
```

Im Sequenzdiagramm, sieht man nochmal den Ablauf der Methode in einer LISTE mit zwei LISTENELEMENTEN (KNOTEN) und einem LISTENELEMENT ABSCHLUSS, in die ein dritter KNOTEN eingefügt wird.



Klassendiagramme der Klassen

Es ergeben sich aus den Vorüberlegungen folgende Klassendiagramme:



Das LISTENELEMENT wird dabei mit einer abstrakten Klasse verwirklicht. Eine abstrakte Klasse wird mit dem Codewort „`abstract class`“ eingeführt. Methoden in einer abstrakten Klasse können als „`abstract`“ definiert werden, was dafür sorgt, dass sie wie bei einer Schnittstelle von der Unterklasse überschrieben werden muss (Theoretisch kann auch die Unterklasse wieder `abstract` gesetzt werden und dann kann die Methode wieder als `abstract` weitergegeben werden).

Beispiel: LISTENELEMENT und KNOTEN

```
public abstract class LISTENELEMENT
{
    abstract LISTENELEMENT HintenEinfuegen(DATENELEMENT d);

    //Weitere Methoden
}

public class KNOTEN extends LISTENELEMENT
{
    …
    LISTENELEMENT HintenEinfuegen(DATENELEMENT d)
    {
        nachfolger = nachfolger.HintenEinfuegen(d);
        return this;
    }
…
}
```

Aufgaben:

1) Kopiere die Vorlage aus dem Klassenordner (Swing_Arztzimmer_06a). Mache parallel deinen alten Stand der allgemeinen LISTE auf.

2) Überlege dir, welche Methoden man...

- … einfach auf die zwei Klassen ABSCHLUSS und KNOTEN aufteilen kann.
- … beim Übertragen in ABSCHLUSS und KNOTEN einen neuen Rückgabetyp bekommen müssen und somit neue Befehle brauchen.
- … welche Methoden nun schon im LISTENELEMENT erscheinen müssen und welche exklusiv im KNOTEN bestehen können.

3) Erstelle geeignete Struktogramme (und wo nötig Sequenzdiagramme), um dir die Funktionsweise der Methoden von ABSCHLUSS und KNOTEN zu veranschaulichen.

4) Erstelle in der Vorlage die neuen Klassen LISTENELEMENT und ABSCHLUSS. Sorge dafür, dass ABSCHLUSS und KNOTEN von LISTENELEMENT erben.

5) Ergänze die Klassen ABSCHLUSS, KNOTEN und LISTENELEMENT um die Methoden der allgemeinen LISTE. Achte darauf, dass alle Methoden in ABSCHLUSS und KNOTEN umgesetzt sind. Teste anschließend ausführlich.

6) Hast du es schon verstanden? Falls ja, dann **versuche** überzeugend zu erklären, warum man für die Umsetzung keine Schnittstelle verwenden kann.