Christian-Albrechts-Universität zu Kiel
Sommersemester 2000
Fortgeschrittenenpraktikum (Java)

Termin: 9. Mai 1998

Pizzas (Aufgabe 1, 2, 4) Die meisten Probleme schien die Aufgaben über die Pizzas gemacht zu haben, insbesondere Aufgabe 5. Zunächst einmal die Einstiegsdefinition. Gefordert war in der ersten Aufgabe, einen Datentyp `Pizza_D` in einer Form zu definieren, daß man Pizzas wie folgt instantiieren kann:

```
Pizza_D p = new Zwiebel(new Olive (new Boden()));                (1)
```

Den entscheidenden Hinweis auf eine mögliche Lösung hoffte ich damit zu geben, indem ich sagte, daß die Pizzas *induktiv* definiert seien: Entweder als Boden, oder eine Pizza entsteht aus einer kleineren Pizza durch hinzufügen eines bestimmten Belages, wie man es an der oberen Zeile sehen kann. Im wesentlichen sollte man sich eine *lineare Liste* vorstellen, deren Knoten aus Instanzen der einzelnen Klasse besteht.

Unabhängig davon: was klar war und was alle richtig gemacht haben, ist die Tatsache, daß die konkreten Pizzas als *konkrete Klassen*, die Pizza als solche als *abstrakte Klasse* definiert wurde.

Was aus Gleichung (1) hervorging, ist der Typ der *Konstruktoren* für die Pizzas: er nimmt eine Pizza und liefert eine neue zurück. Die Schwierigkeit bestand offensichtlich darin, was der Effekt der Konstruktoren sein sollte. Was alle erkannt haben, ist, daß, um Fragen wie "Nur-Zwiebeln" an die Pizzas Stellen zu können, die Pizzaklassen gewisse Dinge gespeichert werden müssen. Die einfachste, natürlichste, und für das weitere Vorgehen brauchbare Lösung ist, sich einfach "den Rest der Pizza" zu merken:

```
class Paprika extends Pizza_D {
  Pizza p;                              // Feld
  Paprika(Pizza_D p) {                  // Konstruktor
    p = _p;                             // Man speicher den ‘‘Rest’’
  };
};
```

Dies ist eine Umsetzung der Aufgabenstellung: "Eine Pizza bestehe aus einem Boden und belegt man eine Pizza mit einem der Beläge (hier `Paprika`, ist es immer noch eine Pizza". Damit hat man eine induktive Definition der Pizzas als eine Art "verzeigerte Liste"; richtige Pointer gibt es zwar in Java nicht, aber, wie in der Vorlesung erklärt, werden was Objekte (und Arrays) betrifft, immer nur Referenzen gespeichert und übergeben.

Manche der abgegebenen Lösungen kann man als bestimmte *Abstraktionen* der oben beschriebenen verzeigerten Liste betrachten, indem man nur für relevant erachtete Informationen speichert. Die Lösungen ziehen Nutzen aus der Einsicht, daß man in Aufgabe 2 von Übung 2 eben nur bestimmte Information von den Pizzas extrahieren will, nämlich die Frage nach "nur-Zwiebeln" oder "vegetarisch". Die spartanischste Lösung (die keiner gebracht hat) wäre demzufolge, einfach sich nur *zwei Bits* zu merken, also boolesche Werte, einen für je einer der Fragestellungen. Daneben gibt es manche Zwischenlösungen: insbesondere wurde vorgeschlagen, sich

- einen Vektor von booleschen Werten, für jeden Belag einen Eintrag, oder
- einen Vektor von Integerwerten

zu merken.

Diese Vorschläge lösen diese Aufgabe, sind aber in mancher Hinsicht unbefriedigend:

1. Sie sind maßgeschneidert für die Aufgabe. Sobald eine Methoden verlangt wird, die die Frage stellt `Zwiebeln_auf_Lamm`, taucht also auf der Pizza eine Schicht Zwiebel über einer Schicht Lamm auf, hilft das bloße Zählen nicht mehr weiter. Natürlich kann man dann auch noch die Reihenfolge irgendwie speichern, aber das Entscheidende ist:
2. merkt man sich nur gewisse Werte, die einen gerade interessant scheinen, hat man im Grunde gar keinen "Datentyp" konstruiert. Eine Analogie zwischen Pizzas und, sagen wir *Container*[1] Eine Pizza kann weitere Pizzas enthalten, analog: ein Container ist ein graphisches Element, welches weiterer Container als Unterstrukturen enthalten kann.
   - Eine Pizza, auf die ich eine neuen Belag hinzufügen, ist etwas anderes wenn ich mir einfach die Anzahl der Zwiebeln. Analog
   - Wenn ich einem Container einen neuen hinzufügte ist etwas anders als würde ich nur zählen, wieviele Unterstrukturen er besitzt. (ich will die Dinge ja am Bildschirm sehen.)

Es ist nicht verwunderlich, daß sich die alternativ-Lösungen für die Erweiterungen nicht sehr eignen, aber das ist genau das, was geübt werden soll: Lösungen, die sich einfach erweitern und anpassen lassen.

Ausdrucken der Pizzas Akzeptiert man die Darstellung der Pizzas als verzeigerte Liste, ist das Ausdrucken einfach. Wie in der Vorlesung und auf dem Übungszettel erwähnt, sucht die Methode `println` nach einer Methode `toString`, die jedes Objekt als Instanz der Unterklassen von `java.lang.Object` besitzt. Diese muß nur geeignet überschrieben werden, für Paprikas beispielsweise[2]

```
..
  public String toString() {
    return "new Paprika" + "(" + p.toString() + ")";
  }
..
```

[1] natürlich kann man andere komplexer aufgebaute Datenstrukturen auch nehmen.

[2] Es würde sogar `return " new Paprika"+ "("+ p + ")";` gehen, denn die Operation `+` versucht in diesem Fall, `toString` an `p` aufzurufen.

Pizza: Aufgabe 5 Nun die Aufgabe, die die meisten Diskussionen ausgelöst hat. Den Sinn habe ich versucht, in den Mails und in den Diskussionen zu klären. Hier also nur die Darstellung der Lösung.

Ziel ist, wie gesagt, die Trennung von dem Datentyp und möglichen Operationen darauf. Die Daten, auf denen gearbeitet werden soll, sind, wie erläutert, eine bestimmte Form von Listenstrukturen. Anstelle daß nun jede einzelne Knotenart der Liste die *Implementierung* je für die in den Aufgaben erforderten Methoden bereithält, lagern wir die Implementierung in eigenen Klassen aus, sodaß beispielsweise der Kode für die Frage "nur-Zwiebeln?" nicht länger über Paprikas, Zwiebeln, Boden, etc. verteilt ist, sondern an einer Stelle *zusammengefaßt*. Natürlich stellen die einzelnen Pizzaklassen immer noch die entsprechenden *Methoden* an ihrer Schnittstelle zur Verfügung, aber anstelle sie selber direkt zu *implementieren*, wenden sie sich mit einem geeigneten Methodenaufruf an die entsprechende Klasse, die die jeweiligen Implementierungen zusammenfaßt.

Für jede der geforderten Methoden gibt es eine eigene Klasse `NurZ_V`, `IstVeg_V`, `Sub_ZdL_V` und `Rem_Knob_V`. Die Instanzen dieser Klassen sind die *Implementierungen der Methoden*. Die Listenelemente müssen auf diese zugreifen können; aus diesem Grund werden die Funktionen in entsprechende *Felder* gespeichert. Da dies für alle Arten von Pizzas uniform geht, stecken wir sie in die abstrakte Klasse `PizzaD` und lassen sie von den Unterklassen erben:

```
public abstract class PizzaD{
  NurZ_V     nurZ_fn    = new NurZ_V();
  Sub_ZdL_V  sub_ZdL_fn = new Sub_ZdL_V();

  abstract boolean nurZ();
  abstract PizzaD  sub_ZdL();

};
```

Die Klasse der Zwiebeln, beispielsweise muß vier Methoden zur Verfügung stellen[3] aber sie implementiert sie nicht selbst, sondern ruft nur die entsprechende Methode an den Instanzen auf, die die Methoden implementieren.

```
public class Zwiebel  extends PizzaD {
  PizzaD p;
  Zwiebel (PizzaD _p) {
    p = _p;
  };

  boolean nurZ() {
    return nurZ_fn.forZwiebel(p);          // Parameter p
  };

  PizzaD sub_ZdL() {
    return sub_ZdL_fn.forZwiebel(p);
  };

  public String toString() {
    return ("new Zwiebel (" + p.toString() + ")");
  };

};
```

Es fehlt noch die Definition der Visitorenklassen selbst. Wie gesagt, sie enthält im wesentlichen nichts anderes als zusammengefaßt den Code, der vorher in den einzelnen Klassen selbst beheimatet war. Hier beispielsweise den Code, der die Methode `nurZ()` für alle Unterklassen von `PizzaD` implementiert. Interessant ist der Fall für Zwiebeln.

```
public class NurZ_V {
  boolean forBoden() {
    return true;
  };
  boolean forPaprika(PizzaD p) {
    return false;
  };
```

[3] Hier sind nur zwei gezeigt, dafür zusätzlich noch die `toString`-Methode.

```
  boolean forZwiebel(PizzaD p) {
return p.nurZ();                  // rek. Aufruf !
  };

  boolean forLamm(PizzaD p) {
    return false;
  };
};
```

Hat man das alles geschluckt, ist der Rest auch nicht mehr schwer, hier die Klasse, die die Substitution bewerkstelligt.

```
public class Sub_ZdL_V {
  PizzaD forBoden() {
    return new Boden();
  };

  PizzaD forPaprika(PizzaD p) {
    return new Paprika (p.sub_ZdL());      // rek. Aufruf
  };
  PizzaD forZwiebel(PizzaD p) {
    return new Lamm(p.sub_ZdL());          // ersetzen
   };

  PizzaD forLamm(PizzaD p) {
    return new Lamm(p.sub_ZdL());
  };
};
```