Aufgabe 4: OO-Techniken

OO-Notation, Eigene Klassen, Abstrakte Klassen

Vorbemerkung und Lernziele

Lernziel

- Ziel dieser Aufgabe ist es **OO-Techniken** der Programmierung zu erlernen
- das ist ziemlich schwierig und dauert lange, denn
  - es gibt viele OO-Techniken für verschiedene Zwecke
  - ihr Potential entfaltet sich dadurch, daß sie alle **wechselseitig** ineinander **geschachtelt** angewendet werden können
  - das erfordert eine **relative** Sichtweise auf der Basis von **Spezifikationen**, nicht auf der Basis von konkrete Implementationen
  - man muß deshalb ständig den persönlichen „Hut“ zwischen **Spezifizierer** und **Implementierer** wechseln
  - das heißt die **Abstraktion durch Spezifikation** ist die **unabdingbare** Voraussetzung für erfolgreiche OO-Programmierung
  - Sie müssen lernen, dieses wechseln des jeweiligen „Huts“ (d.h. der Abstraktionsebene) „**automatisch**“ zu machen
  - Dieser ständige Wechsel muß schließlich im Sub-Sekundentakt **unterhalb** der Ebene des expliziten Nachdenkens erfolgen
  - Das erfordert **sehr viel Übung** (viele Jahre, wie in jedem anderen Beruf (z.B. Pianist) auch)

Vorgehensweise

- um sich auf die OO-Sichtweise eines Problems konzentrieren zu können, transformieren wir **zunächst nur vorhandene Lösungen** in OO-Programme
- Sie müssen nicht neu über den **Algorithmus** zur Lösung der Probleme nachdenken, der bleibt gleich
- Sie müssen nur eine **Transformation** Ihrer Programme in ein anderes **Programmierkalkül** vornehmen
- Das basiert im Wesentlichen auf klassenbasierten automatischen Fallunterscheidungen
- deswegen machen wir zunächst die Transformationen in sehr kleinen Transformationsschritten

Teilaufgabe 1

- Transformation der Wochentage in OO
- dazu gibt es viele Beispiel-Implementationen in den Folien
- diese beschäftigen sich mit der Implementation von Clocks und müssen durch Abstraktion auf die Wochentage umgesetzt werden
- wegen der **Überschaubarkeit** des Wochentagproblems ist dieses ein guter Einstieg

Teilaufgabe 2

- Transformation der Graphikaufgabe in OO
- viel mehr Datentypen und Rekursion
- dazu müssen Sie sich etwas mehr anstrengen

Wochentage in OO

Schritt 1: Transformation in OO-Notation

- Transformieren Sie die bisherigen **globalen Funktionen** in eine äquivalente Version mit Polymorphie.
- Behalten Sie dazu zunächst die Klassendefinitionen mit **def_class** bei, da diese viele Methoden **automatisch** erzeugen.
- Z.B. die Methode **==,** auf der das assert_equal(…) für das Testen beruht.

Vorgehensweise

- Transformieren Sie zunächst die Tests, wenn nötig
- Danach die zu testenden Funktionen und Methoden, wenn nötig
- Immer **abwechselnd** in **kleinen Schritten**
- Nennen Sie die Methoden um, wenn nötig
- Z.B. so

  day_sym_to_day_num -> _. to_day_num

- D.h. diejenigen Namensbestandteile der Funktionen, die ausdrücken, welche Daten konsumiert werden, fallen nun wegen der Polymorphie weg

Transformation der Typprädikate

- Transformieren Sie diejenigen **Typprädikate, die nicht automatisch definiert werden.**.
- Verwenden Sie die **Hook-Technik** mit zwei oder mehr Implementationen (eine davon in Object).

Transformation der Konversionen und Operationen

- Brechen Sie die **klassenbasierten** Fallunterscheidungen durch Polymorphie auf
- D.h die **expliziten** Fallunterscheidungen (if … elsif … elsif … else) werden zu **impliziten** Fallunterscheidungen

Analyse

- Prüfen Sie, wo **duplizierter Code** entstanden ist.
- **Speichern** Sie diese Version in einem Skript.

Schritt 2:
Definition eigener Klassen

- Das bisher verwendete **def_class** erzeugt automatisch Klassen mit diversen Methoden.
- z.B. initialize, Selektoren, to_s, ==, etc.
- Eigene Methoden konnten hinzugefügt werden.
- Jetzt sollen Sie die Klassen **explizit** selbst bauen.
- In den Folien finden Sie nähere Erläuterungen.

Vorgehensweise

- Transformieren Sie **schrittweise** die def_class in eigene Klassen, immer nur eine Klasse zur Zeit.
- Die bisherigen Tests bleiben dabei **unverändert**.
- Aber Sie brauchen einige Tests mehr, denn Sie müssen jetzt auch die neuen **selbstdefinierten** Methoden testen.
- Das gilt besonders für ==.
- Testen Sie == möglichst früh, denn darauf beruhen die Tests mit assert_equal(…).

Analyse

- Prüfen Sie, wo **duplizierter Code** entstanden ist.
- **Speichern** Sie diese Version in einem Skript.

Schritt 3: Abstrakte Klassen

Bauen Sie eine abstrakte Klasse für **Day** und **faktorisieren** Sie gemeinsame Methoden heraus.

Definieren Sie **abstrakte Methoden** in den abstrakten Klassen, die eine Fehlermeldung generieren, falls in den Unterklassen eine geforderte Implementation fehlt.

Die Klassenhierarchie sollte dann so aussehen
(abstrakte Klassen in rot)

Day
- **DayNum**
- **DaySym**

Vorgehensweise

- Die Tests bleiben **unverändert**.
- Sie sollten die abstrakte Klasse am Anfang **leer** lassen, und Schritt für Schritt Methoden in die Klasse verschieben.
- Dabei nach jeder einzelnen Transformation **testen**.
- Diese **Refaktorisierungstransformation** nennt man auch "**push up**".
- In der Vorlesung besprechen wir weitere **Techniken zur Refaktorisierung**.

Erweiterung der Funktionalität

Diese Erweiterungen sind jeweils nur wenige Zeilen Code.

Definition arithmetischer Operatoren

- Definieren Sie arithmetische Operatoren (succ, pred, +, -)
- dann können wir Berechnungen auf Days als normale arithmetische Ausdrücke schreiben, das ging bisher nicht
- Beipiele:

  DaySym[:So].succ #=> DaySym[:Mo]

  DaySym[:So] + 3 #=> DaySym[:Mi]

Graphik in OO

Schritte 1 und 2

- Siehe Transformation der Days
- Die Vorgehensweise ist gleich.

Schritt 3: Abstrakte Klassen

Eine minimale Klassenhierarchie könnte so aussehen (abstrakte Klassen in rot)

GraphObj[]
  Point[]
    Point2d[x,y]
  Shape[]
      Range2d[x_range,y_range]
      Union1d[left,right]
      Union2d[left,right]

- Es kann sehr sinnvoll sein, **weitere abstrakte Klassen** einzuführen, um Code herausfaktorisieren zu können.
- Welche sollen Sie selbst herausfinden

Erweiterungen

Erweiterung um Differenz von Shapes

- Neben der Vereinigung von Shapes ist es auch hilfreich Differenzen bilden zu können (Mengendifferenz).
- D.h. wir ziehen von einem Shape etwas ab. Damit kann man **Löcher** in Shapes schneiden.
- Sie brauchen dazu die Klassen **Diff1d** und **Diff2d**
- Sie müssen dafür dann natürlich **bounds etc.** neu implementieren.
- Bei den bounds machen wir es uns einfach.
- Die **Boundingbox einer Differenz** soll einfach der linke Operand sein.
- Das ist zwar etwas pessimistisch, aber einfach zu implementieren. Sonst wird es zu kompliziert.

Definition arithmetischer Operatoren

Wir bilden Beschreibungen von Shapes, indem wir **geschachtelte Ausdrücke** bilden.

Diese Ausdrücke lassen sich einfacher schreiben und lesen, wenn wir normale arithmetische Operatoren (+,-) verwenden können.

Dabei steht + für Vereinigung und - für Differenz.

Sei

```
R1  = Range2d[0..2,3..5]
R2  = Range2d[-1..1,-1..1]
R3  = Range2d[1..1,1..1];
```

Dann sind folgende Ausdrücke gleichbedeutend

```
Diff2d[Union2d[R1,R2],R3]
```

und der viel kürzere Ausdruck

```
R1 + R2 - R3
```

Freiwillig: Zwei Repräsentationen für Rechtecke

Eine wichtige Forderung besteht darin, alternative austauschbare Implementationen anbieten zu können.

Wir wollen dieses an einer weiteren Implementation der Rechtecke durchspielen.

- Bisher waren die Rechtecke durch zwei Ranges definiert.
- Alternativ kann man auch die linke untere und die rechte obere Ecke nehmen
- Z.B. Klasse **Rect2d[ll,ur]**
- dabei steht ll für lower left und ur für upper right
- Beide Implementationen müssen sich funktional gleich verhalten.
- Z.B. muß jede Implementation die **Selektoren** des anderen auch implementieren.
- die Objekterzeugung muß auch gleich möglich sein
- das heißt, Sie müssen das initialize modifizieren
- Weiterhin bietet sich eine weitere abstrakte Klasse **Rect** an, in die die Gemeinsamkeiten herausfaktorisiert werden können.