Skript Informatik II
Sommersemester 2007

SS 2007

Dozenten:
Prof. Dr. H. Klaeren /
Eric Knauel

Mitschrieb von
Rouven Walter

Lizenz

Das Werk „Informatik 2“ von Rouven Walter steht unter einer Creative Commons Namensnennung-Nicht-kommerziell-Weitergabe unter gleichen Bedingungen 3.0 Deutschland Lizenz. Eine Zusammenfassung der Lizenz ist unter http://creativecommons.org/licenses/by-nc-sa/3.0/de/ einsehbar. Der vollständige rechtsverbindliche Lizenzvertrag kann eingesehen werden unter http://creativecommons.org/licenses/by-nc-sa/3.0/de/legalcode. Alternativ kann ein Brief an folgende Adresse geschrieben werden: Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Vorwort

Dieser Mitschrieb entstand während meiner Nachbearbeitung zur Informatik II Vorlesung im Sommersemester 2007 bei Prof. Dr. H. Klaeren an der Eberhards-Karl-Universität Tübingen.
Ich erhebe keinen keinen Anspruch auf Vollständigkeit oder Richtigkeit. Bei Unklarheiten zum Inhalt empfehle ich daher, sich an die jeweiligen Dozenten/Tutoren zu wenden.

Über Verbesserungsvorschläge oder sonstige Anregungen, würde ich mich über eine Email an: *RouvenWalter (AT) web.de* freuen.

Danksagung

Mein Dank geht an Nasira Sindhu, Karin Schaller und Jonas Ströbele, die mich auf Fehler im Skript aufmerksam gemacht haben.

Links

Universität Tübingen:
http://www.uni-tuebingen.de/

Wilhelm-Schickard-Institut für Informatik:
http://www.informatik.uni-tuebingen.de/

Website zur Vorlesung 'Informatik II':
http://www-pu.informatik.uni-tuebingen.de/info-ii-07/

Inhaltsverzeichnis

Lizenz....iii
Vorwort....iii
Danksagung....iii
Links....iii
Übersicht über den Vorlesungsinhalt....1
  Thema....1
  Plan....1
  Ziele....1
  Literatur....1
1.Arten von Daten....2
  1.1.Einfache/atomare/primitive Sorten von Daten....2
  1.2.Klassen....2
  1.3.Eingeschachtelte Klassen....4
  1.4.Vereinigung von Klassen....5
    1.4.1.Typen und Klassen....7
  1.5.Vereinigung, Selbst-Referenzen und Wechselseitige Referenz....7
  1.6.Initialisierung von Feldern....10
  1.7.Example Klassen....11
  Fallstudie "Krieg der Welten" (Teil 1)....11
2.Funktionale Methoden....12
  2.1.Berechnung mit primitiven Typen....13
  2.2.Methoden definieren....14
  2.3.Methoden testen....17
  2.4.Bedingte Auswertung....18
  2.5.Methoden in Klassendiagrammen....18
  2.6.Methoden für geschachtelte Daten....19
  2.7.Methoden für gemischte Daten....20
  2.8.Methoden-Dispatch....23
  2.9.Methoden für Daten mit wechselseitigen Referenzen....23
  2.10.Bibliotheken....26
  2.11.Fortsetzung Fallstudie "Krieg der Welten" (Teil 2)....26
3.Mit Klassen abstrahieren....27
  3.1.Abstrakte Methoden, abstrakte Klassen....29
  3.2.Methoden in der Klassenhierarchie nach oben ziehen....30
  3.3.Nachträgliche Vereinigung von Klasse....32
  3.4.Unterklassen ableiten....34
  3.5.Bibliotheken für Grafik-Animationen....35
  3.6.Subtypen und Kompatibilität....35
  3.7.Gleichheit....37
    Mini-KA für equals-Methoden....38
  3.8.Sichtbarkeit und Zustand....38
    Anleitung für PS:....39
    Überladene (private) Konstruktoren....39
  3.9.Zustandsänderung und Zuweisung....39
  3.10.Lokale Variablen....40
  3.11.Schlüsselwort static....42

3.12.Schleifen....42
3.13.Arrays....43
3.14.Parametrische Polymorphie (mit Java Generics)....44
3.15.Ausnahmebehandlung mit Exceptions....45
4.Grafische Benutzeroberflächen in Java....46
4.1.Reaktivität mit Callbacks....48
4.2.Model-View-Controller-Architecture (MVC)....48
4.3.Observer-Pattern....49
5.Systematischer Umgang mit Fehlern....49
Traffic-Prinzip....50
Literatur zur Fehlersuche....51
5.1.Debugger....51

Übersicht über den Vorlesungsinhalt

Thema

Objektorientierte Programmierung und Java

Objekt-Orientierte Programmierung (OOP)

– Vielzahl von Definitionen

– *Klasse* und *Objekt* bevorzugte Abstraktionsmittel

– verbreitet seit Beginn der 80er Jahre

– Vielzahl von OO-Sprachen:

- Simula-67
- Smalltalk
- Ada 95
- Object C
- C++
- Java
- C#

Plan

– Schrittweise Einführung in Java

– Sprachlevel, die Aspekte und Konstrukte ausblenden

– Entwicklungsumgebung mit Sprachleveln: ProfessorJ (alias DrScheme)

– Ziel: vollständiger Sprachumfang von Java

Ziele

1) Syntax + Semantik von Java

2) Know-How zur sinnvollen Verwendung der OO-Abstraktionsmöglichkeiten

Literatur

- Peter Sestoft: Java Precisely, MIT Press, 2. Auflage, 2005
- Joshua Bloch: Effective Java, Addison-Wesley, 1.Auflage, 2001
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification, Addison-Wesley, 3.Auflage, 2005

1. Arten von Daten

Vorgehen mit Konstruktionsanleitung.

1) Solides Verständnis für das Problem entwickeln (Kurzbeschreibung, Beispiel)

2) Entwickle Repräsentation für die involvierten Daten

3) ...

→ Repräsentation der Daten in Java?

1.1. Einfache/atomare/primitive Sorten von Daten

```
Int -1, 0, 42
```

ganze Zahlen mit Vorzeichen, Wertebereich von $-a$ bis $+a$, $a=2^{31}=2147483648$

→ später mehr

```
double -1.0, 0.42, 3E8
```

Gleit-/Fließkommazahlen, Wertebereich? → später mehr

```
boolean false, true
```

Wahrheitswerte

```
String "abc"
```

Zeichenkette

1.2. Klassen

Häufig: Relevante Informationen setzen sich aus mehreren Teilen zusammen

Beispiel:

...Programm zur Rechnungsstelle bei einem Kaffeeimporteur

...Rechnungsposition: Sorte, Preis/Pfund, Menge

```
In Scheme:
; Eine Kaffee-Rechnungsposition ist ein Wert
;     (make-coffee kind price weight)
; mit kind String; price, weight Zahl
(define-record-procedures coffee
     make-coffee coffee?
```

```
     (coffee-kind coffee-price coffee-weight))
>(make-coffe "Kona" 1595 100)
```

In Java repräsentiert man diese Sorte Daten als *Klasse*.

```
class Coffee {
     String kind;          \
     int price;             > Felder
     int weight;           /
     ↑ Typ, ↑ Name
     Coffee(String kind, int price, int weight) {
          this.kind = kind;          \
          this.price = price;         > Konstruktor
          this.weight = weight;      /
     }
}
```

Eine Klasse ist ein Entwurfsmuster für die Erzeugung ("Instantiierung") von *Objekten*.

Eine Klasse definiert eine neue Sorte (Typ) Daten. Alle Objekte, die Instanz dieser Klasse sind, sind von dieser Sorte. Der Aufruf des *Konstruktors* einer Klasse erzeugt eine neue Instanz: ein Objekt.

Syntax für Konstruktor (vorläufig):

```
Klassenname(typ1 feld1, ..., typn feldn) {
     this.feld1 = feld1;
               ...
     this.feldn = feldn;
}
```

Für alle Felder der Klasse.

Syntax für Aufruf des Konstruktors:

Bsp.:

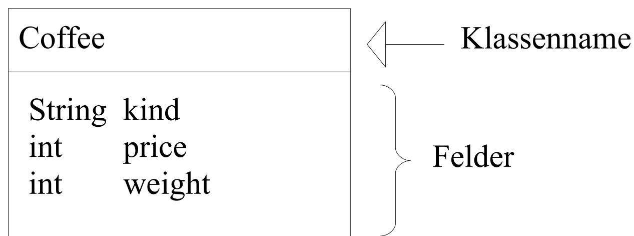
```
new Coffee("Kona", 1595, 100)
```

Allgemein:

```
new Klassenname(arg1, ..., argn)
```

Klassendiagramme

In der Regel enthält ein Programm viele Klassen. Daher stellt man Klassen übersichtlich dar:



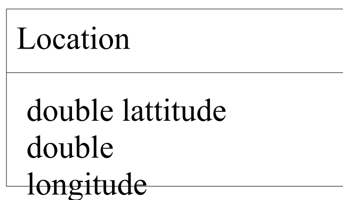
Bsp.:

Problem:

...Navigationssystem, welches GPS-Daten geliefert bekommt. Längen-/Breitengrad.

lattitude 48,53, longitude 9,03

lattitude 53,54, longitude 10



1.3. Eingeschachtelte Klassen

Beispiel:

Entwickle ein Trainingslogbuch für einen Langstreckenläufer.

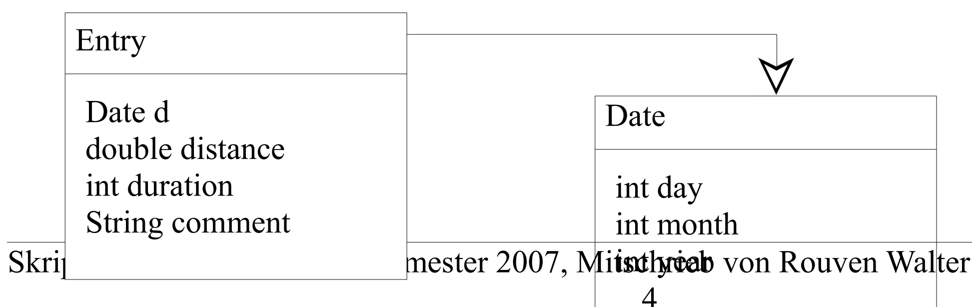
Tägliche Einträge:

Datum, Distanz, Dauer, Kommentar,

5.Juni 2006, 5,4km, 27min, erschöpft

Beobachtung:

Zusammengesetzte Daten. Dauer ist int, Distanz ist double, Kommentar ist String, Datum ist *wiederum zusammengesetzt*.



A ⟶ B:

Klasse A verwendet (hat Felder des Typs) B.

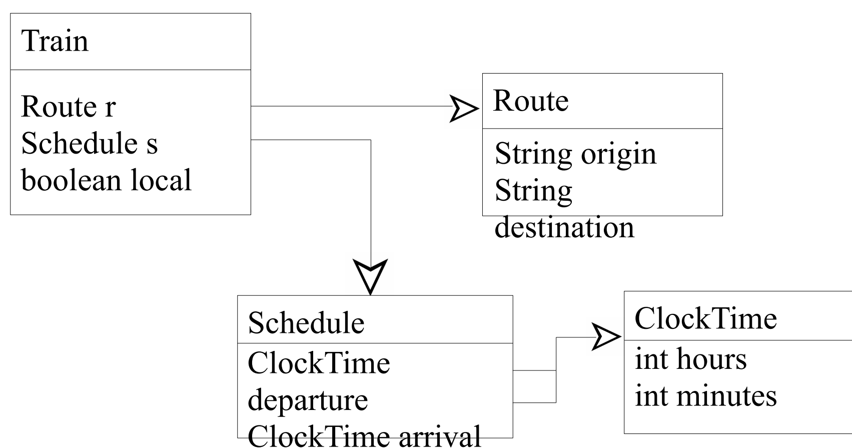
Der Pfeil setzt direkt am Klassenkasten an.

Weiteres Beispiel:

Programm zur Reiseplanung von Zugfahrten.

Informationen:

Nah-/Fernverkehr, Route (Start, Ziel), Fahrplan (Abfahrt, Ankunft)



Definition in REPL:

```
Typ var-name = Ausdruck;
```

Begriff:

```
Route r1 = new Route("Stuttgart", "Hamburg");
int x = 5;
```

1.4. Vereinigung von Klassen

Zum Zugbeispiel aus 1.3.

– Arten von Zügen werden über Boolean-Feld unterschieden

– Was passiert, wenn weitere Arten von Zügen hinzukommen?

=> Problem: Repräsentation wird kompliziert

Besser: Je Art von Zug eine Klasse.

=> Problem:

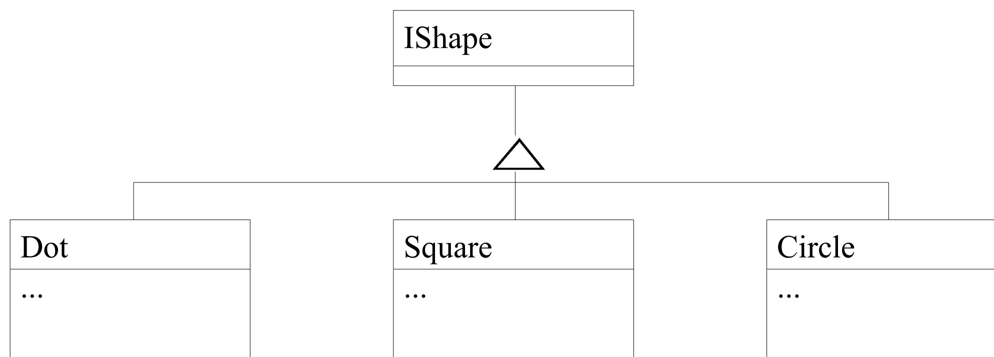
Jede Klasse ist eine neue Sorte.
Was ist dann ein "Zug"?

=> Gemischte Daten

Beispiel aus Informatik 1 (Präsenzübung 4):

Zeichenprogramm mit 3 Sorten geometrischer Figuren:

Punkte, Quadrate, Kreise, angeordnet auf kartesischem Koordinatensystem.



Ishape ist ein *Interface*.

Interface:

Ein Interface ist ein Entwurfsmuster für Klassen.

Klassen, die ein Interface i *implementieren* entsprechen diesem Entwurfsmuster und weisen alle Eigenschaften des Interfaces i auf.

Eigenschaften?

→ "Klasse c implementiert Interface"

→ weitere Eigenschaften später

Syntax für Interface-Definition:

```
interface IName { }
```

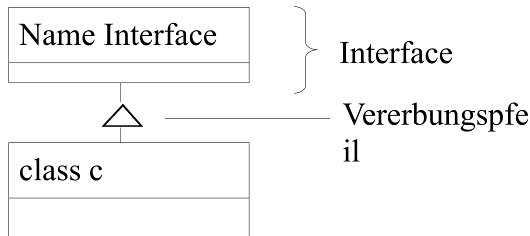
Syntax für Klassendefinition mit Interface:

```
class Name implements Iname { }
```

Beispiel:

```
class Dot implements Ishape { }
```

Im Klassendiagramm werden Interfaces und implements-Beziehungen dokumentiert:



1.4.1. Typen und Klassen

Bei der Definition von Feldern müssen wir einen *Typ* abgeben. Was genau ist ein Typ?

– Primitive Typen (int, double, ...)

– Klassen und Interfaces

Enthält ein Programm `Ty inst;` oder `Ty inst = new Cls(...)`, dann heißt das:

1) Die Variable inst hat den Typ Ty

2) Die Variable inst steht für eine Instanz von Cls

3) Das Programm ist nur korrekt, wenn Cls Ty ist oder Cls Ty implementiert. Anderenfalls: Fehler.

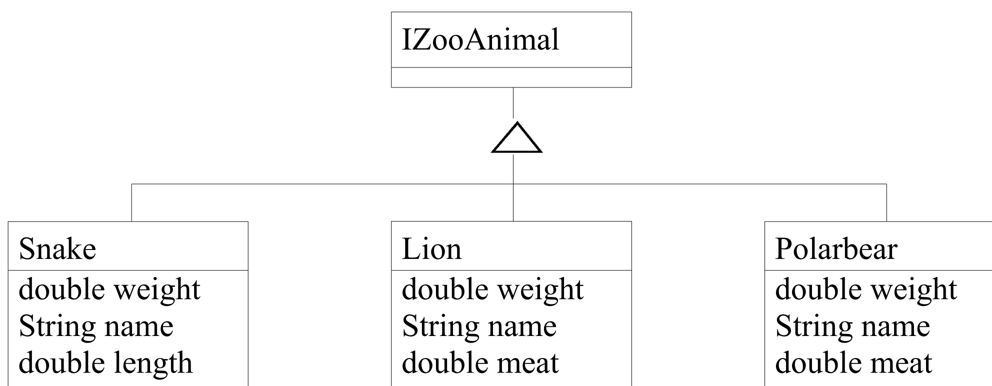
Weiteres Beispiel zu Interfaces:

Tierpfleger, der Tiere eines Zoos verwaltet (Löwen, Schlangen, Eisbären). Jedes Tier hat Namen, Gewicht. Schlangen: Länge. Für die anderen Arten: Futtermenge.

Bsp.:

1) "Leo" wiegt 150kg und frisst 2,5kg/Tag

2) "Kaa" wiegt 20kg und ist 1,80m lang



1.5. Vereinigung, Selbst-Referenzen und Wechselseitige Referenz

Bisher haben wir Informationen aus dem Problem durch feste Anzahl von Objekten repräsentiert:

– eine Rechnungsposition (Coffee)

– ein Eintrag im Trainingslogbuch (Entry)

→ Repräsentation für Sammlungen von Informationen

Bekanntes Problem:

Entwickle Trainingslogbuch für den Langstreckenläufer mit täglichen Eintragungen:

Datum, Distanz, Dauer, Kommentar.

Bsp.:

5.Juni 2006, 5,3km, 27min, "erschöpft"

Mehrere Einträge repräsentieren (in Scheme):

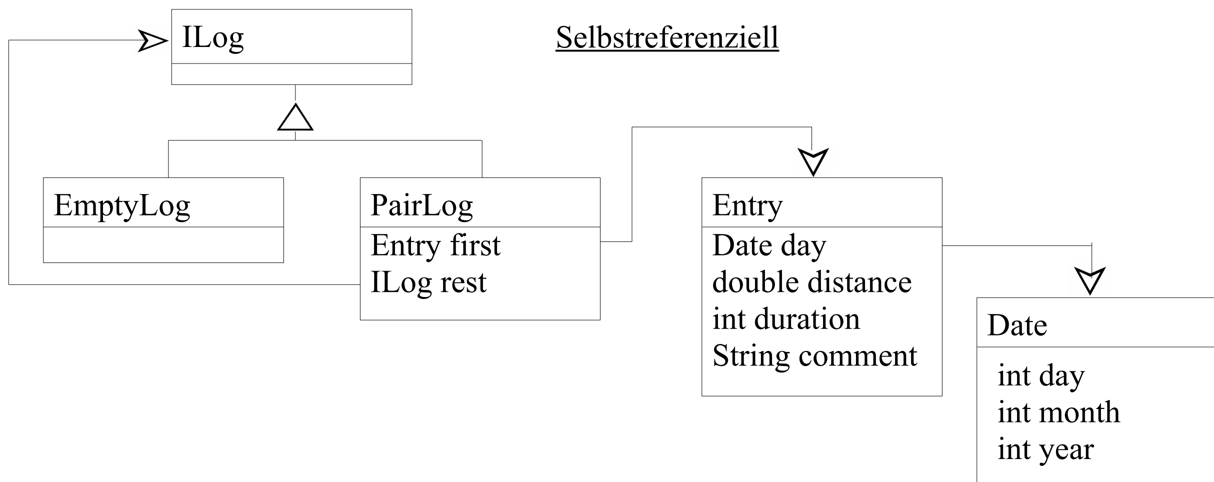
```
; Ein Eintrag ist entweder:
;- die leere Liste empty
;- ein Paar (make-pair Entry Log)
```

→ Gemischte Daten; KA für Vereinigung von Klassen

1) Interface Ilog für alle Logs

2) Klasse EmptyLog für das leere Log

3) Klasse PairLog; repräsentiert Log, welches existierendes Log erweitert



Weiteres Beispiel:

Betrachtetes Programm für geometrische Figuren mit zusätzlichem Feature: Eine Kombination von Formen durch übereinanderlegen ergibt eine neue Form.

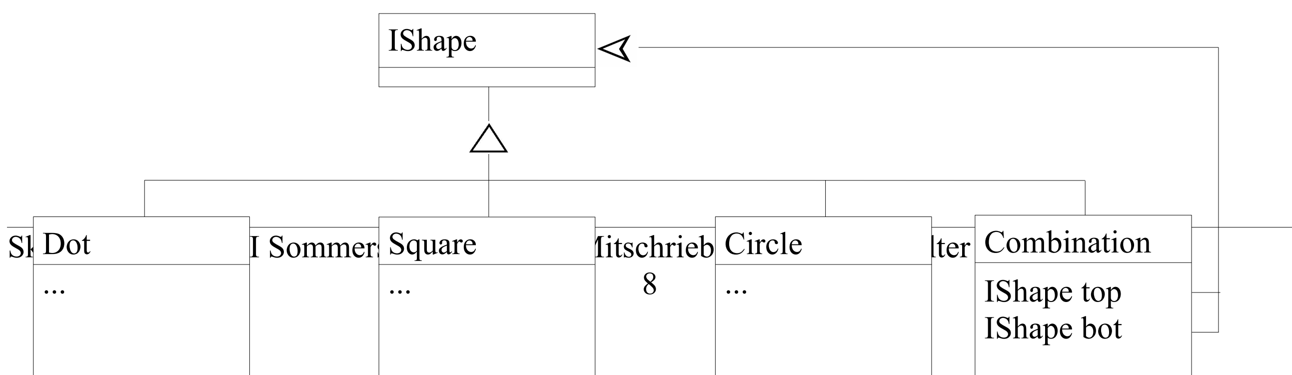
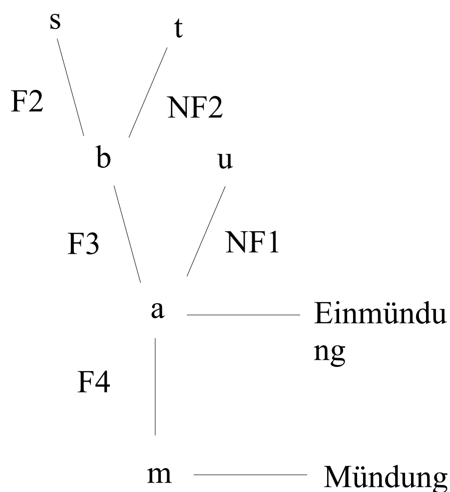


Diagramm ist auch selbstreferenziell, aber keine Liste!

Weiteres (komplexeres) Beispiel:

Das Umweltbundesamt überwacht Wasserqualität der Flüsse. Ein Flusssystem besteht aus einem Fluß, Nebenflüssen und den Nebenflüssen der Nebenflüsse usw. und aus einer Quelle und einer Mündung (Ort).



Mündung besteht aus:

- Ort
- Fluß

Fluß *ist*:

- Einmündung
- Quelle

Einmündung besteht aus:

- Ort
- linker Fluß
- rechter Fluß

Ort besteht aus:

– Koordinaten

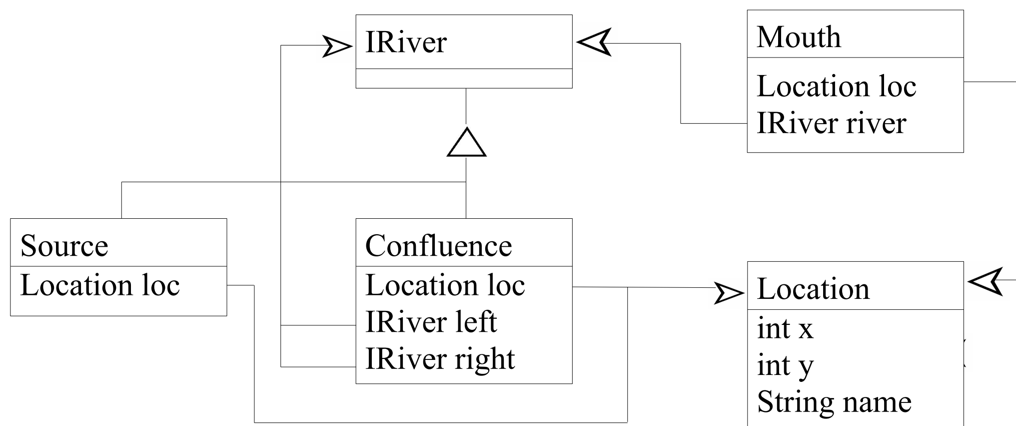
– Name

Quelle besteht aus:

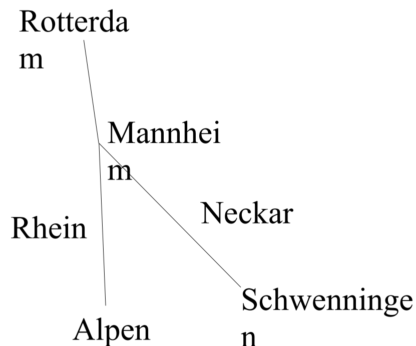
– Ort

– (Fluß)

Klassendiagramm:



Bsp.:



1.6. Initialisierung von Feldern

Bisher:

Alle Felder eines Objekts wurden durch die als Argumente für den Konstruktor übergebenen Werte initialisiert.

Auch Felder, die für alle Instanzen einer Klasse gleich sind.

```
Class UnitCircle {
     Cartesian pos;
     int radius;
```

Repräsentator für Einheitskreis

```
    UnitCircle(Cartesian pos, int radius) {
        this.pos = pos;
        this.radius = radius;
    }
}
```

Macht nur Sinn, wenn Konstruktoraufrufe immer Radius 1 enthalten.

1. Bessere Lösung:

Anderer Konstruktor.

```
UnitCircle(Cartesian pos) {
    this.pos = pos;
    this.radius = 1;
}
```

2. Bessere Lösung:

Feld mit fester Initialisierung.

```
Class UnitCircle {
    Cartesian pos;
    int radius = 1;

    UnitCircle(Cartesian pos) {
        this.pos = pos;
    }
}
```

1.7. Example Klassen

Klassen, deren Name auf "Example" endet werden in ProfessorJ besonders behandelt:

Bei Klick auf "Start" werden Instanzen dieser Klasse erzeugt.

Zweck: Testfälle schreiben

Wichtig: Konstruktor darf keine Argumente haben.

Fallstudie "Krieg der Welten" (Teil 1)

Aufgabe:

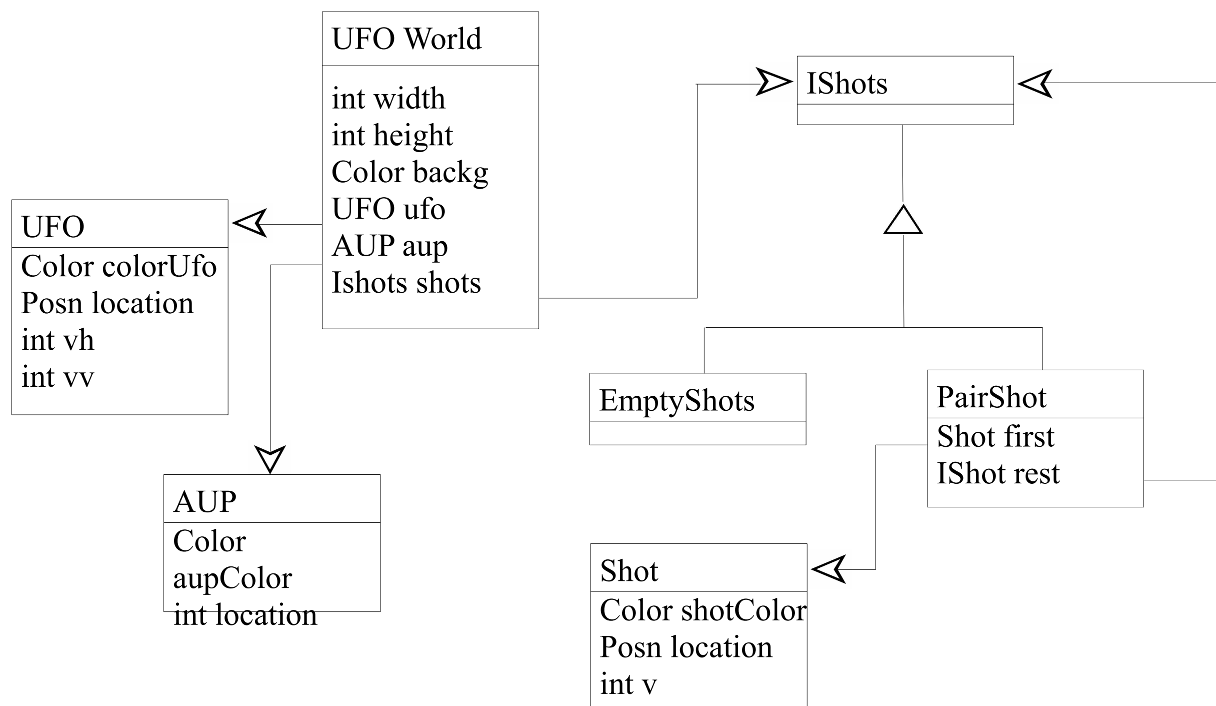
Entwickle das Spiel "Krieg der Welten":

Ein UFO sinkt vom Himmel herab. Der Spieler bewegt am Boden eine Anti-UFO-Plattform (AUP) mit der Schüsse auf das UFO abgefeuert werden können. Landet das UFO, ist das Spiel verloren; wird es abgeschossen, ist das Spiel gewonnen.

Was muss repräsentiert werden?

1) Die komplette Welt des Spiels mit allen enthaltenen Objekten (Zeichenfläche 200x500 Pixel).
2) Ein UFO. Es erscheint am zufälligen Platz in der Welt und sinkt. x, y-Koordinaten, Größe, Sinkgeschwindigkeit, Geschwindigkeit für horizontale Bewegung.
3) Ein AUP. Reagiert auf Tastendrücke, x-Position.
4) Abgefeuerte Schüsse. Liste von Geschossen.
5) Ein Geschoss im Flug (nach oben), x-y Koordinaten, Geschwindigkeit.

Klassendiagramm:



Beobachtung:

Die Definition von Color und Posn fehlen!

Dieser Code befindet sich in einer *Bibliothek* (in Java *package* genannt). Einbinden des Codes mit:

```
import geometry.*;
import colors.*;
```

} Am Anfang der Datei

Fügt die Klassen Posn (kartesische Koordinate) und Colors (Red, Green, Yellow, usw.) hinzu.

2. Funktionale Methoden

Methoden sind grob betrachtet etwas ähnliches wie die Prozeduren: Beide konsumieren Daten und produzieren Daten.

Methoden sind immer mit Klassen verknüpft:

Eine Instanz der Klasse ist *Primär-Argument* beim Aufruf einer Methode.

Daher: Aufruf einer Methode (einer Instanz) vs. Prozeduraufruf.

Beispiel:

(Prozeduraufruf in Scheme)

```
> (string-length "hello world")
> 11
```

In Java:

```
"hello world".length()
```

zusätzliche Argumente (→ `()`)

Primär-Argument (`"hello world"`), Name der Methode (`length`)

```
"hello".concat("world") vs (string-append "hello" "world")
```

Allgemein:

```
eObject.methodName(expression, ... )
```

eObject ist ein Ausdruck, der zu einem Objekt auswertet. Häufig einfach Variablenname:

```
name.length()
```

Ausdruck:

```
"hello".concat("world").length()
```

String-Objekt "helloworld" (`"hello".concat("world")`)

2.1. Berechnung mit primitiven Typen

– Infix-Notation für Ausdrücke:

Scheme: (+ a b)

Java: a + b

– Ausdrücke müssen nicht vollständig geklammert sein.

→ Welche Operanden gehören zu einem Operator?

Lösung: Operatoren binden unterschiedlich stark. (15 Präzedenzstufen)

	Symbol	Stelligkeit	Eingabtypen	Ergebnistyp	Beispiel	Bedeutung
Stufe 1	`!`	unär	boolean	boolean	`!(x < 0)`	logisches nicht
Stufe 2	`*` `/`	binär	Zahl, Zahl	Zahl	`x * 2` `x / 2`	Multiplikation Division
Stufe 3	`+` `-`	binär	Zahl, Zahl	Zahl	`x + 2` `x - 2`	Addition Subtraktion
Stufe 4	`<, <=, >, >=, ==`	binär	Zahl, Zahl	boolean	`x < 2` `x <= 2`	Kleiner, Kleiner-gleich, ...
Stufe 5	`&&` `||`	binär	boolean, boolean	boolean	`x && (y < 0)`	logisches und, logisches oder

Der Methodenaufruf "." steht auf Stufe 0

Beispiel:

```
      (and (not (< x 0)) (< x 10))
→     !(x < 0) && (x < 10)
```

Klammer bei `(x < 0)` notwendig, da `!` stärker bindet als `<` .

Klammer bei `(x < 10)` nicht notwendig, dadurch wird der Code aber leichter lesbar.

Beachte:

Runde Klammern in Java diesen dazu eine bestimmte Auswertungsreihenfolge zu forcieren oder deutlich zu machen.

`((x < 10))` ≡ `(x < 10)`   (Java)

`(< x 10)` ist in Schme zulässig, aber `((< x 10))` nicht.

Nachtrag:

String ist eine Klasse und kein primitiver Typ.

Wichtige Methoden von String:

Methode	Eingabetyp	Ergebnistyp	Bedeutung
length	-	int	Anzahl der Zeichen
concat	String	String	zusammenhängen
toLowerCase toUpperCase	-	String	in kleine/große Buchstaben umwandeln

2.2. Methoden definieren

Bekanntes Problem:

Programm zur Rechnungsstellung bei einem Kaffee-Importeur. Rechnungsposition enthält Sorte,

Preis/Pfund, Menge.

Gesucht: Betrag für Rechnungsposition.

```
; Eine Kaffeerechnungsposition ist ein Wert
; (make-coffee kind price weight)
; wobei ...
(define-record-procedures coffee
  ...)

; compute cost for this invoice line item
; cost: Coffee -> number
(define (cost a-coffee)
     (...))
```

laut entsprechender KA

Jetzt KA für zusammengesetzte Daten zur Verfügung

```
; ...
; ...
```

unverändert

```
(define (cost a-coffee)
     ... (coffee-kind a-coffee) ...
     ... (coffee-price a-coffee) ...
     ... (coffee-weight a-coffee) ...)
```

KA: Selektoren verwenden, um Werte aus dem Record zu extrahieren

Zwischenbemerkung:

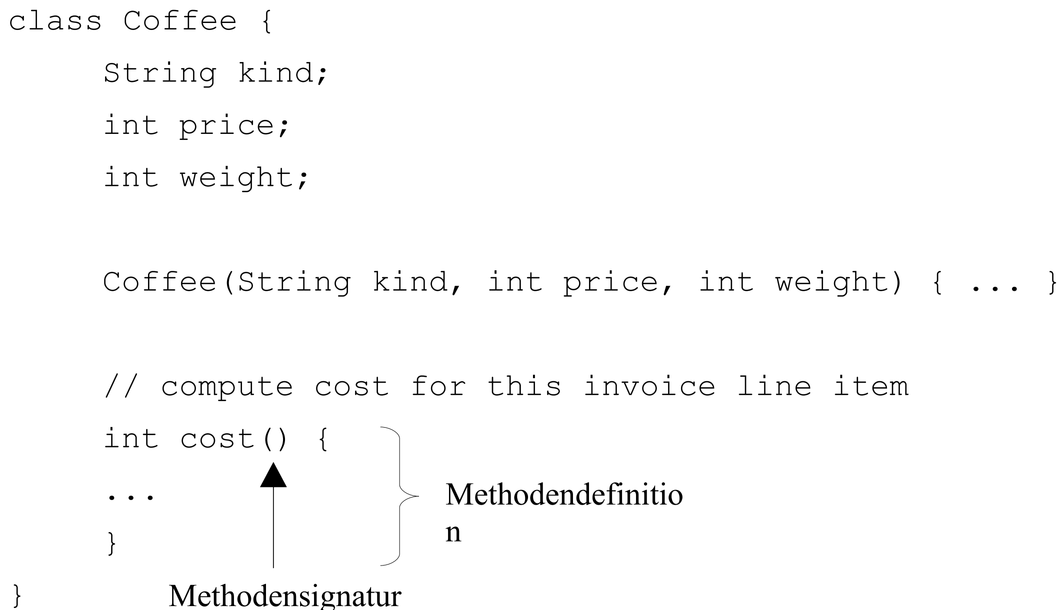
```
(define f
     (lambda (x)          ≡          (define (f x)
          ... ))                          ... )
```

Fertige Prozedur:

```
(define cost
     (lambda (a-coffee)
          (* (coffee-price a-coffee)
             (coffe-weight a-coffee))))
```

In Java:

Zusammgesetzt → Klasse definieren



```
class Coffee {
      String kind;
      int price;
      int weight;

      Coffee(String kind, int price, int weight) { ... }

      // compute cost for this invoice line item
      int cost() {
      ...
      }
}
```


– Kurzbeschreibung schreiben

– Der Vertrag (Scheme Kommentar) wird teil der Methoden-Signatur:

`; cost: Coffee -> int` vs `int cost()`

So wird festgelegt, dass `cost` einen int-Wert produziert (*Rückgabetyp*).

– Wo ist das Argument *a-coffee* geblieben? Wie legt man fest für welche Instanz `cost` berechnet wird? Die Instanz ist das Primär-Argument und taucht deshalb nicht in der Argumentliste auf.

Methodenaufruf:

```
<Ausdruck>.cost()
```

→ Ausdruck muss eine Instanz von Coffee auswerten

Diese Instanz ist an dem Standardnamen *this* gebunden.

Aufruf der Methode:

```
> new Coffee("Kona", 1595, 100).cost()
```

Primär-Argument ("this" in cost())

```
> Coffee c = new Coffee("Kona", 1595, 100);
> c.cost();
```

}

Das Schlüsselwort *return* steht vor einem Ausdruck, der der Rückgabewert einer Methode ist.

```
return <Ausdruck>
```

Weiteres Beispiel:

Methode, die herausfindet, ob eine Rechnungsposition teurer ist als ein gegebener Betrag.

```
// to determine whether this coffee sale costs more than amount
boolean moreCents(int amount) {
     return this.price > amount;
}
```

Weiteres Beispiel:

Wiegt eine Rechnungsposition mehr als eine andere?

```
// determine whether this coffee sale is lighter than that sale
boolean lighterThan(Coffee that) {
     ...this.kind...that.kind...
     ...this.price...that.price...       → return this.weight <
     ...this.weight...that.weight...       that.weight
}
```

2.3. Methoden testen

Die Example Klassen (vgl. 1.7) können auch Tests für Methoden enthalten. Die Tests sind Methoden der Form:

```
boolean testName() {
     ....
}
```

wichtig: Methodenname mit Präfix "test"

Rückgabe: true, falls Test erfolgreich war, anderenfalls false.

Vergleiche von Objekten und primitiven Typen werden mit dem *check/expect*-Ausdruck durchgeführt.

```
boolean testCost() {
  return check new Coffee("Kona", 1595, 100).cost() expect 159500;
}
```

Test-Ausdruck: `new Coffee("Kona", 1595, 100).cost()`

Ausdruck: erwartetes Ergebnis: `159500`

Ausdruck: `check new Coffee("Kona", 1595, 100).cost() expect 159500`

Für Fließkommazahlen muss diese Form von check/expect verwendet werden:

```
check Ausdruck₁ expect Ausdruck₂ within Ausdruck₃
```

Bedeutet: Ist der Wert von $Ausdruck_1$ gleich Wert $Ausdruck_2$ ± Wert von $Ausdruck_3$.

Es können auch mehrere Tests in einer Testmethode durchgeführt werden:

```
return (check ... expect ...) && (check ... expect ...) ...;
```

2.4. Bedingte Auswertung

Die bedingte Auswertung erfolt mit *if/else*.

If/else ist ein Statement (in Scheme ist if ein Ausdruck).

```
if(<condition>)
     <Konsequente>;     \
else                      Statements
     <Alternative>;     /
```

Condition muss zu einem boolean-Wert auswerten. Ist dieser Wert true, wird die Konsequente ausgewertet, anderenfalls die Alternative.

Beispiel:

Sparbuch mit Zinsen in Abhängigkeit vom Guthaben.

0 - 50.000 €	→	2%
50.000 € - 100.000 €	→	2,25%
> 100.000 €	→	2,5%

```
class Account {
     String owner;
     int amount;
     ...
     double rate() {
          if ((0 <= this.amount) && (this.amount <= 50000))
```

```
                return 2.0;
            else
                if((50000 <= this.amount) && (this.amount <= 100000))
                    return 2.25;
                else
                    return 2.5;
    }
}
```

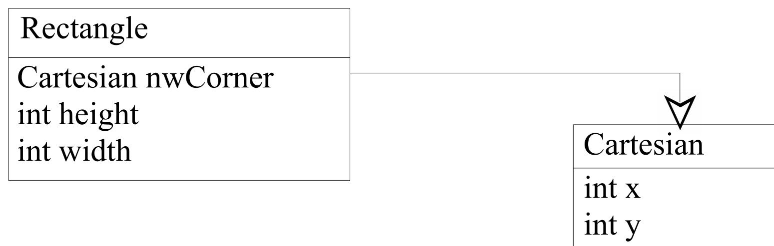
2.5. Methoden in Klassendiagrammen

Methoden werden auch im Klassendiagramm dokumentiert.

2.6. Methoden für geschachtelte Daten

Beispiel:

Rechtecke auf einer Zeichenfläche. Abstand der Rechtecke zum Ursprung berechnen.



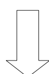
→ Eigentlich nach zwei Methoden gefragt:

1) distanceOrigin für Rectangle

2) distanceOrigin für Cartesian

Cartesian:

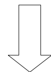
```
double distance() {
    ...this.x...this.y...
}
```



```
double distanceOrigin() {
     return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

Rectangle:

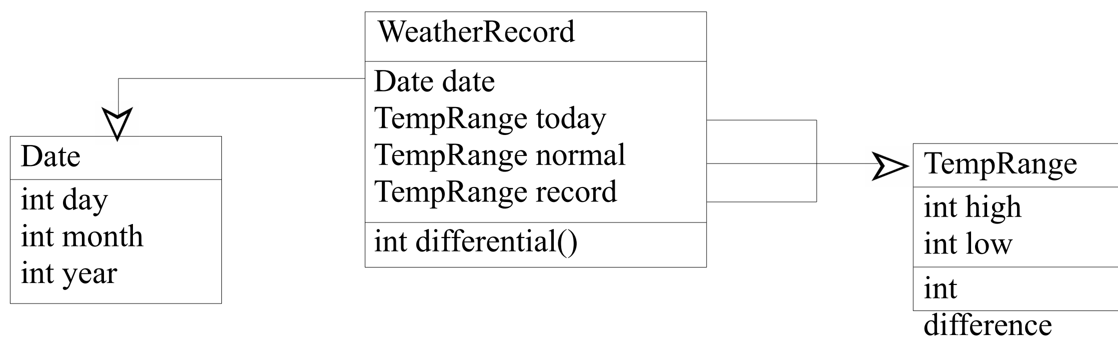
```
double distanceOrigin() {
     ...this.nwCorner...
}
```



```
double distanceOrigin() {
     return this.nwCorner.distanceOrigin();
}
```

Weiteres Beispiel:

WeatherRecord vom Übungsblatt 1.



Aufgabe: Temperaturschwankung eines Tages berechnen.

KA 8.

1) Kurzbeschreibung

```
// compute the difference between todays high and low
```

2) Datenanalyse

– geschachtelte Daten liegen vor

– eingeschachtelte Daten (aus TempRange) werden gebraucht.

→ KA 8, um Methode für TempRange zu schreiben

1) `// compute difference of this TempRange`

2) Datenanalyse

3) Schablone

```
int difference() {
```

```
        return this.high - this.low;
    }
```

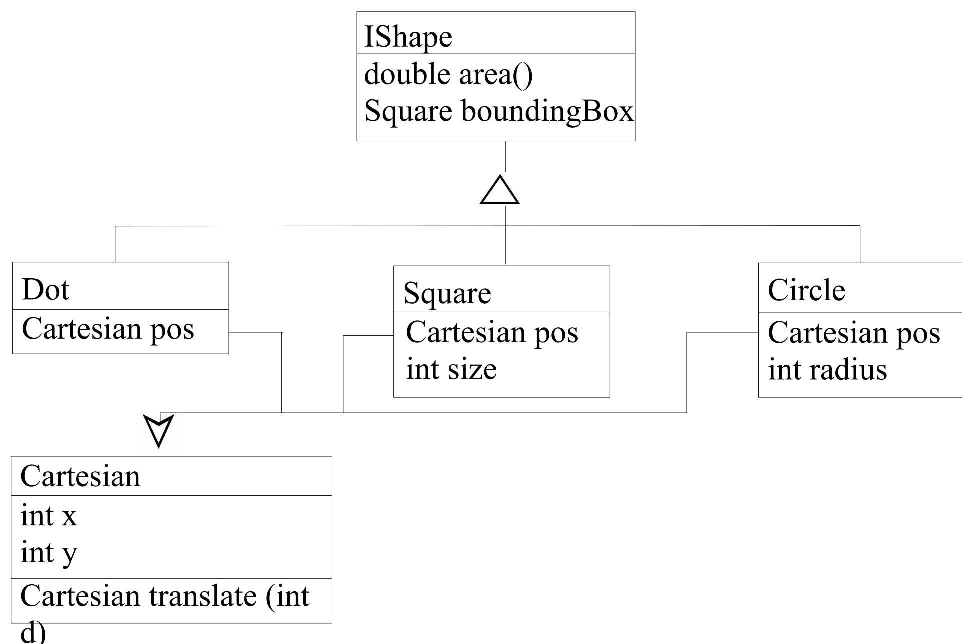
4) Testen

Fortsetzung KA 8 für Methode auf WeatherRecord.

3) Schablone

```
int differential() {
    return this.today.difference();
}
```

2.7. Methoden für gemischte Daten



Aufgabe: Methode, die den Flächeninhalt einer Form berechnet

- Methode muss für Dot, Square, Circle (alle Klassen, die Ishape implementieren) funktionieren
- jeweils andere Formel für Flächeninhalt
- Methoden-Signatur `double area()`

Lösung:

1) Die Methode wird im Interface IShape beschrieben.

Es gilt:

Interfaces sind Entwurfsmuster für Klassen. Alle Methoden, die im Interface aufgeführt sind, müssen von allen Klassen, die dieses Interface implementieren, mit der Interface gegebenen

Signatur definiert werden.

Sonst: Fehler.

2) Methoden für Dot, Square und Circle definieren

Dot:

```
double area() {
     return 0.0;
}
```

Square:

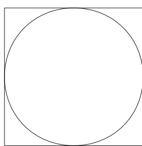
```
double area() {
     return this.size * this.size;
}
```

Circle:

```
double area() {
     return Math.PI * this.radius * this.radius;
}
```

Weiteres Beispiel:

Aufgabe: Umschließendes Rechteck (hier immer Quadrat) einer Figur.



"bounding box"

```
> new Circle (new Cartesian(0, 0), 20).boundingBox()
→ new Square(new Cartesian(-20, 20), 40)
```

Nach KA 8:

1) Kurzbeschreibung

```
// compute the bounding box for this shape
```

2) Datenanalyse

– gemischte Daten (Ishape)

→ KA 10

– Methoden-Signatur zum Interface IShape hinzufügen.

Dot:

```
// ...
Square boundingBox() {
	return new Square(this.pos, 1);
}
```

Square:

```
// ...
Square boundingBox() {
	return this;
}
```

Circle:

– geschachtelte Daten, die auch zur Berechnung benötigt werden

Schablone:

```
Square boundingBox() {
	...this.pos.translate(...)...
}
```

2.8. Methoden-Dispatch

Das Programm für geometrische Figuren enthält drei unterschiedliche Methoden gleichen Namens.

area() für Dot, Square und Circle.

Wie wird festgestellt, welche Methode aufgerufen werden muss?

→ polymorpher Methoden-Dispatch

1) Auswerten des Ausdrucks für das Primär-Argument (PA)

2) Ist der Wert ein Objekt? Nein → Fehler

3) Feststellen von welcher Klasse das Objekt im PA einer Instanz ist

4) Such in dieser Klasse die entsprechende Methodendefinition

5) Eigentlicher Aufruf

Stark vereinfachtes Modell zum Verständnis des Methodenaufrufs; technische Umsetzung ist *sehr viel* komplizierter!

Vgl. Scheme

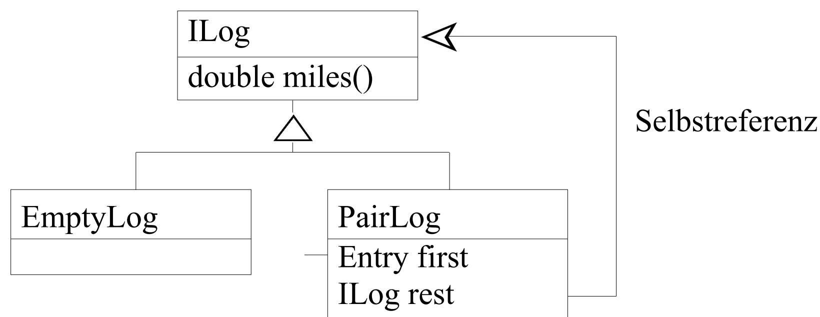
```
(define area
	(lambda (shape)
		(cond
```

Dieser Mechanismus ist in Java direkt in der Programmiersprache eingebaut

```
((dot? shape) (dot-area shape))
((square? shape) (square-area shape))
(...))))
```

2.9. Methoden für Daten mit wechselseitigen Referenzen

Beispiel: Trainingslogbuch



Methode, um die Gesamtlänge der Laufstrecke zu berechnen.

```
// to compute the total number of miles in this log
    double miles()
```

KA für gemischte Daten.

Schablone:

EmptyLog:

```
    double miles() {
        ...
    }
```

PairLog:

```
    double miles() {
        ...this.first...this.rest...
    }
```

ILog
→geschachtelte Daten

fertig stellen

EntryLog:

```
    double miles() {
```

```
        return 0.0;
    }
```

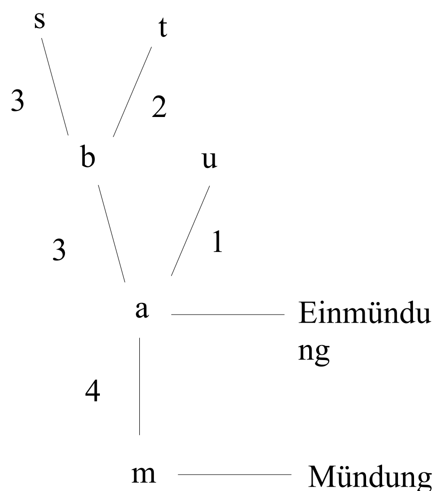
PairLog:

```
    double miles() {
        return this.first.distance + this.rest.miles();
    }
```

Sieht aus wie list-sum in Scheme:

```
(define list-sum
    (lambda (lst)
        (cond
            ((empty? lst) 0)
            ((pair? lst)
                (+ (first lst) (list-sum (rest lst)))))))
```

Beispiel: Flusssystem (vom 24.04.2007)



Aufgabe:

1) Länge der einzelnen Segmente erfassen

2) Gesamtlänge des Flusssystems berechnen

Zu 1):

Füge `int length` in Klassen `Source` und `Confluence` ein.

Zu 2):

Methode für die Gesamtlänge schreiben.

– Arbeitet auf IRiver → Vereinigung von Klassen

– IRiver um Methodensignatur erweitern

```
int completeLength()
```

– Methodendefinitionen einfügen

Source:

```
int computeLength() {
     return this.length;
}
```

Confluence:

```
int computeLength() {
     return this.length +
             this.left.computeLength() +
             this.right.computerLength();
}
```

2.10. Bibliotheken

Einbinden der Bibliotheken

```
import geometry.*;
import colors.*;
import draw.*;
```

} Einfache Prof J – spezifische Bibliotheken zum zeichnen

am Anfang des Programmes.

Gezeichnet wird auf einer Zeichenfläche (Canvas).

Canvas erzeugen: `new Canvas(Breite, Höhe)`

(Breite, Höhe: int)

Methode `boolean show()`

→ Canvas sichtbar machen (neues Fenster)

Methode `boolean close()`

→ Canvas-Fenster schließen

Methoden zum Zeichnen:

```
boolean drawCircle(Posn p, int r, Color c)
```

→ Zeichne unausgefüllten Kreis an Position p mit Radius r und Farbe c

```
boolean drawDisk(Posn p, int r, Color c)
```

→ Zeichne gefüllten Kreis an Position p mit Radius r und Farbe c

```
boolean drawLine(Posn p0, Posn p1, Color c)
```

→ Zeichne Linie von p0 nach p1

```
boolean drawRect(Posn p, int width, int height, Color c)
```

→ Rechteck mit Ecke links-oben an p zeichnen

```
boolean drawString(Posn p, String s)
```

→ Text s and Position p zeichnen

2.11. *Fortsetzung Fallstudie "Krieg der Welten" (Teil 2)*

Bisher: Repräsentation der Daten

Heute: AUP vervollständigen

1) Steuerung durch Benutzer

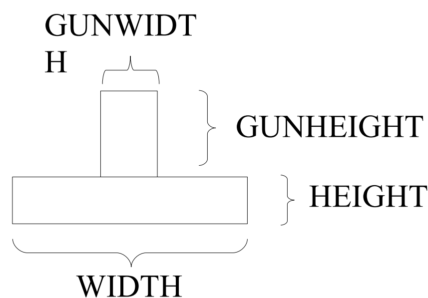
2) Auf Zeichenfläche zeichnen

Zu 1):

Implementierung der Steuerung per Tastendruck kommt später (Controller), erstmal nur Aktualisierung des Modells (und View).

Füge Methode hinzu:

`AUP move(String directions, UFOWorld w)` } aktualisiert Modell



3. Mit Klassen abstrahieren

Beispiel IShapes: Die Klassen, die IShape implementieren, sind sich ähnlich.

(Feld location, Verpflichtung IShape zu implementieren).

Weiter gedacht: Ähnliche Methoden werden mehrfach an verschiedenen Stellen im Programm implementiert anstatt über die Unterschiede zu abstrahieren.

→ Eventuelle Fehler werden somit reproduziert

Lösung: OO-Sprachen: Vererbung.

Die Vererbungsrelation wird bei der Klassendefinition angegeben.

```
class A extends B {
      ...
}
```

Terminologie:

B ist *Superklasse* von A /

A ist eine *Subklasse* von B /

A verfeinert / erbt von / ist abgeleitet von B.

Die Felder und Methoden der Superklasse sind in der Subklasse verfügbar.

→ Gemeinsamkeiten zweier Klassen können in gemeinsame Superklasse "hochgezogen" werden.

Beispiel:

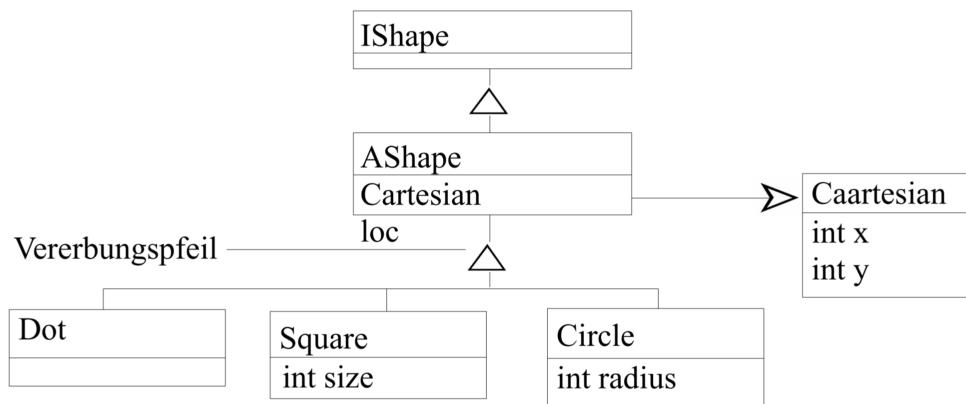
```
class AShape implements IShape {
      Cartesian loc;
      AShape(Cartesian loc) {
            this.loc = loc;
      }
}
```

gemeinsame Superklasse

Subklassen:

```
class Dot extends AShape {
      ...
}
class Square extends AShape {
      ...
}
class Circle extends AShape {
      ...
}
```

Im Klassendiagramm:



Erben von einem Interface ("implements")

→ Erben der Implementierungsverpflichtungen

Erben von einer Klasse ("extends")

→ Erben der Felder, Methoden und Implementierungsverpflichtungen

Beispiel:

Square hat daher zwei Felder

- `int size` (Klassendefinition Square)
- `Cartesian loc` (geerbt von AShape)

Konstruktoren werden in Java generell nicht vererbt.

Daher muss Square beide Felder (geerbte, definierte) initialisieren.

Konstruktor für Square:

```
Square(Cartesian loc, int size) {
      this.loc = loc;
      this.size = size;
}
```

Alternativ:

```
Square(Cartesian loc, int size) {
      super(loc);          // Aufruf des Superkonstruktors (Konstruktor von AShape)
      this.size = size;
}
```

Zusammenhang zwischen Klassen/Interfaces aus:

1) Interface IShape repräsentiert alle geometrischen Figuren.
2) AShape repräsentiert die Gemeinsamkeiten der geometrischen Figuren. Implmentiert IShape.
3) Dot, Square, Circle verfeinern AShape. Erben Felder (und Methoden) und die Implementierungsverpflichtung von IShape.
4) Anhand der Konstruktion lässt sich nicht erkennen, wieviele Felder jede Klasse hat.

3.1. Abstrakte Methoden, abstrakte Klassen

Methoden für geometrische Figuren

- `double area()`
- `double distanceOrigin()`
- `Square boundingBox()`

Methoden standen im Interface IShape

→ Da AShape IShape implementiert, musste AShape eine Defintion für die Methoden aus dem Interface enthalten.
→ Macht z.B. für area() gar keinen Sinn drei unterschiedliche Berechnungen, keine

Gemeinsamkeiten

Lösung:
abstrakte Methoden einführen.

```
    abstract double area();      // Klasse AShape
```

Schlüsselwort (abstract) — Methodensignatur (double area();)

Die Subklassen müssen die abstrakten Methoden implementieren.
Definieren oder Weiterreichung einer Implementierungsverpflichtung.

Macht es Sinn Instanzen von AShape zu erzeugen?
→ Nein, was soll passieren, wenn abstrakte Methode aufgerufen wird?
→ Enthält eine Klasse abstrakte Methoden, so wird die ganze Klasse abstrakt (und kann nicht mehr instanziert werden). Kann aber weiterhin Konstruktor enthalten.

```
abstract class <name> {
    ...
}
```

3.2. Methoden in der Klassenhierarchie nach oben ziehen

Beispiel:

```
boolean larger(IShape s) {
    return this.area > s.area();
}
```

Offensichtlich der gleiche Code für alle drei Formen.

→ Hebe larger() in der Klassenhierarchie an, so dass Square, Dot, Circle diese Methode haben können.

```
abstract class AShape implements IShape {
    Cartesian loc;

    abstract double area();
    boolean larger(IShape s) {
        return this.area() > s.area();
    }
}
```

Hier:
- Dot, Square, Circle erben Methode larger()
- larger() in abstrakter Klasse definiert und verwendet wieder abstrakte Methode

Methoden müssen nicht identisch sein, um in der Klassenhierarchie angehoben zu werden.
Beispiel:
Methode ist für die meisten Varianten einer Vereinigung von Klassen gleich, aber nicht für alle.

Dann Methode in gemeinsame Superklasse anheben und in den Klassen, die eine eigene Definition brauchen, überschreiben.

Beispiel:
`distanceOrigin()`
– für Dot, Square identisch
– für Circle andere Bezeichnung

```
abstract class AShape implements IShappe {
     Cartesian loc;
     ...

     double distanceOrigin() {
          return this.loc.distanceOrigin();
     }
}
```

Überschreiben in Circle:
```
class Circle extends AShape {
     int radius;
     ...

     double distanceOrigin() {
          return this.loc.distanceOrigin() - this.radius;
     }
}
```

Aufruf von Supermethoden:
Beispiel: Methode distanceOrigin() für geometrische Figuren
Letzter Stand:
– Methode verschoben in abstrakte Superklasse AShape
– Dot, Square erben Methode
– Circle überschreibt geerbte Methode mit einer eigenen Version:
```
double distanceOrigin() {
     return this.loc.distanceOrigin – this.radius();
}
```
Hier: Komplett neuer Code in der überschrieben Methode

Häufig so:
– Gemeinsamkeiten in Superklassen
– Spezialisierung in die Unterklasse

} gesamter benötigter Code

Notwendig, den Code aus der Superklasse in der überschriebenen Methode zu nutzen.
→ Aufruf der *Supermethode*

Beispiel:
```
double distanceOrigin() {
     return super.distanceOrigin() - this.radius;
```
Die Methode distanceOrigin() der Superklasse AShape wird aufgerufen.

}

Nützlich in Situationen, in denen die Supermethode und die zu schreibende Methode der Unterklasse ähnliches tun.

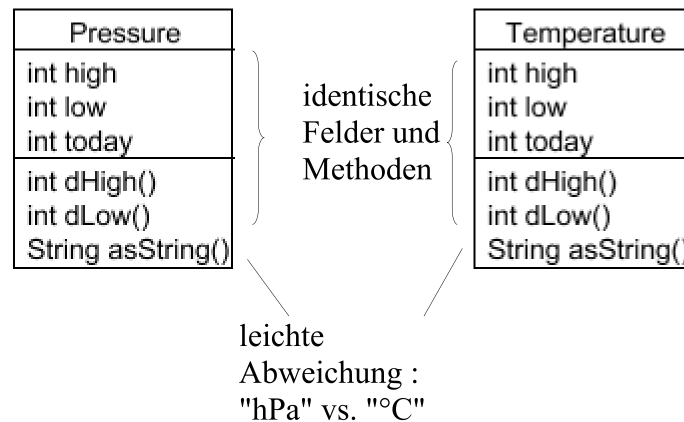
3.3. Nachträgliche Vereinigung von Klasse

Häufige Situation:

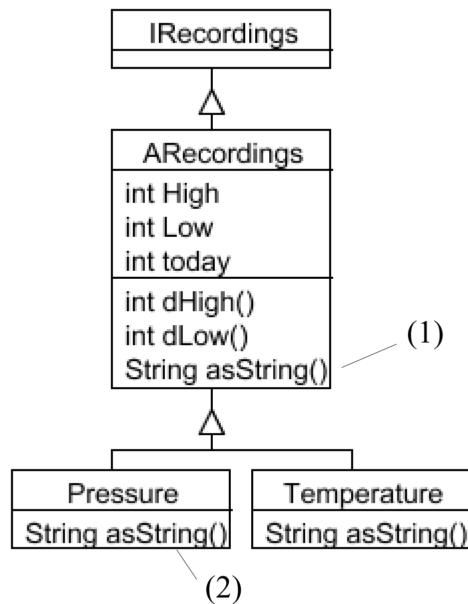
Es stellt sich erst spät(er) in der Entwicklungsphase heraus, dass zwei Klassen sich ähneln, aber nicht Varianten einer Vereinigung sind. (geänderte Anforderungen).

→ Unbedingt erforderlich: Gemeinsamkeiten in eine gemeinsame Superklasse hochziehen und gegebenfalls Vereinigung von Klasse bilden

Beispiel: Wetterdaten erfassen



Variante 1:



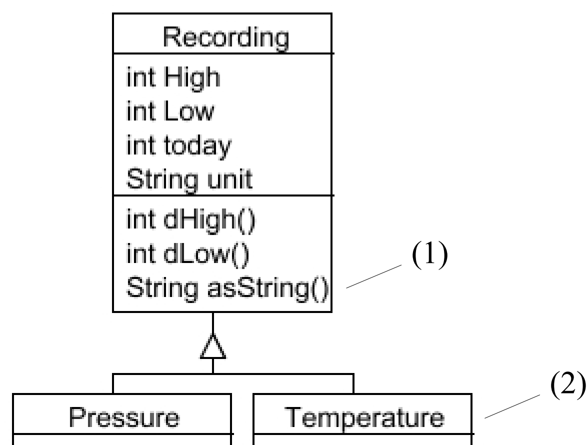
(1)

```
String asString() {
     return String.valueOf(this.high).concat("-").concat(
                String.valueOf(this.low);
}
```

(2)

```
String asString() {
     return super.asString().concat("hPa");
}
```

Variante 2:



(1)

```
String asString() {
     return String.valueOf(this.high).concat("-"). concat(
               String.valueOf(this.unit);
}
```

(2)

Konstruktor:

```
Temperature(int high, int low, int today) {
     super(high, low, today "°C");
}
```

→ Verhalten sich beide Programme gleich? (Aspekt: Hinzufügen weiterer Klassen)

a) Variante 1 *verlässt* sich darauf, dass der Programmierer weiss, dass asString() überschrieben werden muss.

   Variante 2 *zwingt* den Programmierer die Maßeinheit anzugeben (Konstruktor der Superklasse.

b) Nachteil Variante 2: Wo Temperatur und Pressure-Instanzen erwartet werden können auch Recording-Instanzen auftauchen:

   Recording r;

   steht für Temperatur, Pressure oder Recording-Instanzen

   → Es sollte keine Instanzen von Recording geben!

Lösung: Variante 3

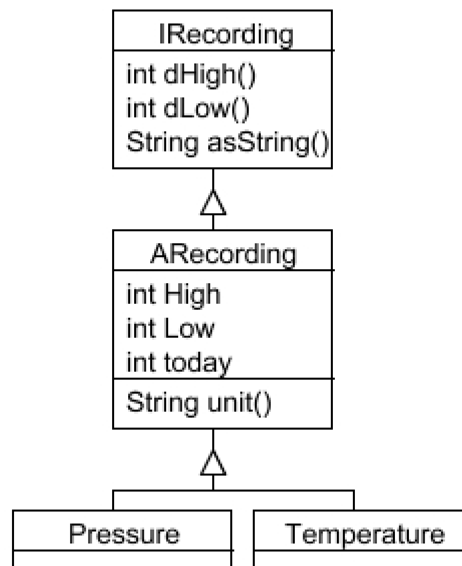
```
abstract class ARecording implements IRecording {
     int high;
     int low;
     int today;
     int dHigh() {...}
     int dLow() {...}
     String asStrign() {
          return String.valueOf(this.high).concat("-").concat(
                    String.valueOf(this.low).concat(this.unit());
     }
     abstract unit();
```

```
}
```

– Keine Instanzen von Recording mehr: Nur noch sinnvolle Instanzen von IRecording

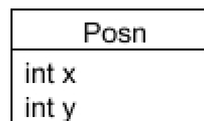
– Verpflichtung Maßeinheit: abstrakte Methode unit()

Klassendiagramm:

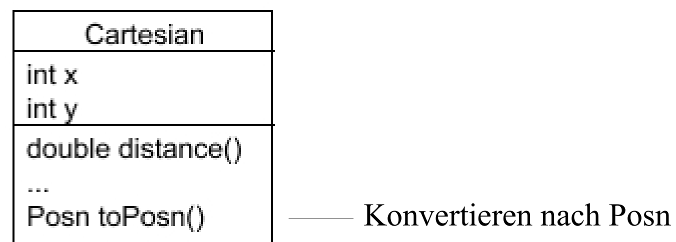


3.4. Unterklassen ableiten

Die Klasse Posn aus dem draw-Package besteht aus 2 Feldern:

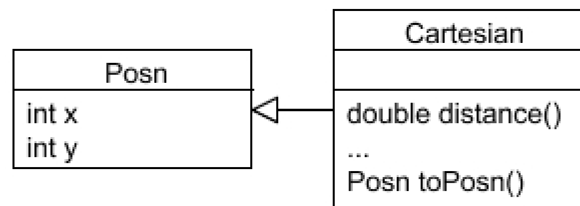


Unsere Methoden, die auf Koordinaten arbeiten, waren in einer Klasse Cartesian.



Bessere Lösung:

Posn als Superklasse von Cartesian. Cartesian fügt Methoden hinzu.



Jede Instanz einer Subklasse hat auch den Typ der Superklasse. Anstelle einer Posn-Instanz kann also eine Cartesian-Instanz angegeben werden.

3.5. Bibliotheken für Grafik-Animationen

Häufig erwarten die Hersteller von Bibliotheken, dass der Benutzer Klassen/Methoden implementiert.

```
abstract class World {
  boolean bigBang(int width, int height, double s);
  abstract World onTick();
  abstract World onKeyEvent(String k);
  abstract boolean draw();
  World endOfWorld(String s);
}
```

3.6. Subtypen und Kompatibilität

t1 kann Subtyp von t2 sein (t2 ist ein Supertyp von t1), geschrieben: t1 <: t2.

→ Werte von Typ t1 können überall verwendet werden, wo Wert von Typ t2 erwartet wird.

Subtyp-Beziehung:

- reflexiv: Für alle t : t <: t
- transitiv: t1 <: t2 und t2 <: t3 => t1 <: t3
- Für primitive Typen t1, t2:
  Wenn es eine Widening-Konversion (W, L) von t1 nach t2 gibt => t1 <: t2

	nach			
von	int	long	float	double
int	W	W	L	W
long	C	W	L	L
float	C	C	W	W

double	C	C	C	W

L: Konversion mit Präzisionsverlust

W: Verlustfreie Widening-Konversion

C: Narrowing-Konversion

- Wenn t1, t2 Klassen sind:
  t1 <: t2, wenn t1 eine Subklasse von t2 ist
- Wenn t1, t2 Interfaces sind:
  t1 <: t2, wenn t1 Subinterface von t2 ist
- Wenn t1 Klasse und t2 Interface:
  t1 <: t2, wenn t1 (eine Subklasse von einer Klasse ist, die) t2 implementiert (oder ein Subinterface von t2)
- Jeder Referenztyp ist ein Subtyp der vordefinierten Klasse *Object*

Automatische Konversion von primitiven Typen erfolgt nur, wenn diese verlustfrei ist

Beispiele:

```
double d = 5; // statt 5.0
long l = 42; // statt 42L
int i = 4.2; // Fehlermeldung
```

TypeCasts konvertieren primitive Typen explizit.

```
(t) l
```

wobei:

`l` : Ausdruck

`t` : Typ zu dem der Wert des Ausdrucks konvertiert wird

Eventuell: Verlust bei Präzision

```
int i = (int) 4.2;      => i = 4
int j = (int) (4.0 + 0.2);       => i = 4
```

Die Subtyp-Beziehung zwischen Referenztypen kann zur Laufzeit durch *instanceof-Operator* geprüft werden.

```
e instanceof t
```

wobei:

`e` : Ausdruck

`t` : Typ

liefert true, wenn der Typ des Wertes von e ein Subtyp von t ist.

```
> new Circle(new Posn(10, 20)) instanceof IShape
> true
> new Circle(new Posn(10, 20)) instanceof Circle
> true
> new Circle(new Posn(10, 20)) instanceof Square
> false
```

Referenztypen können ebenfalls mit Typecasts konvertiert werden, wenn sie in Subtyp-Beziehung stehen.

3.7. Gleichheit

a) Intensionale Gleichheit (Objekt-Identität).

Operator e1 == e2 (equal? in Scheme)

liefert true, wenn e1 und e2 zu exakt demselben Objekt auswerten.

```
> new Object() == new Object()
> false
```
(immer false!)

b) Extensionale Gleichheit

Alle Klassen erben implizit von Object u.a. die Methode

```
boolean equals(Object o)
```

Diese Methode ist für extensionale Gleichheit vorgesehen, muss dafür aber entsprechend überschrieben werden. (Vorsicht: Geerbte Methode verwendet ==)

Eigenschaften von equals():

- reflexiv:
  x.equals(x) => true
- symmetrisch:
  x.equals(y) ≡ y.equals(x)
- transitiv:
  x.equals(y) und y.equals(z) => x.equals(z)

– konstant:
x.equals(y) liefert dasselbe Ergebnis, so lange x, y unverändert bleiben

Überschriebene Methode *muss* diese Eigenschaften haben!

Mini-KA für equals-Methoden

1) Prüfen, ob `this == o`, ggf. true zurück
2) Typ von `o` mit `instanceof` prüfen
3) Typcast zum korrekten Typ
4) Vergleich der Felder

3.8. Sichtbarkeit und Zustand

Zentrales Prinzip: *Geheimnisprinzip*
→ Trennung von Schnittstellen und Implementierung
D.h.:
– Nur Methoden der Schnittstellen sollen benutzbar sein
– Hilfsmethoden und Zustand verstecken

Lösung in Java: *Privacy Specification (PS)*
Vor allen Methoden, Feldern und Konstruktoren steht eine PS:
– *private*: nur innerhalb derselben Klasse nutzbar
– *public*: von beliebiger Klasse nutzbar
– *protected*: nur innerhalb derselben Klasse und allen Subklassen nutzbar

Beispiel:
```
abstract class World {
  protected Canvas theCanvas = ...;

  public boolean bigBang(int ...) {...}
  public abstract boolean draw() {...}
}
```

Anleitung für PS:

1) Im Zweifel private verwenden

2) Methoden, die in einem Interface stehen müssen public sein
3) Wenn eine (nicht verwandte) Klasse eine Methode verwenden muss: public evtl. bessere Lösung Neues Interface
4) Nur von Subklassen benutzte Felder/Methoden; protected
5) Ein Feld ist public, wenn eine nicht-verwandte Klasse darauf zugreifen muss (lesend!)
6) Konstruktoren in abstrakten Klassen sind meistens protected
7) Konstruktoren sind private, es sei denn andere Klassen können diesen Konstruktor mit sinnvollen Argumenten aufrufen

Vor der Ausführung:

Prüfen, ob alle PS beachtet werden

→ sonst Fehler (analog zu Typcheck)

Überladene (private) Konstruktoren

- Mehrere Konstruktoren für eine Klasse (Überladung)
- Unterschiedliche Signaturen
- Auswahl (beim Aufruf) erfolgt passend zu der Zahl und Typen der Argumente
- Nur Konstruktoren, die für andere Klassen gedacht sind public deklarieren
- Aufruf eines Konstruktors aus einem anderen Konstruktor:

```
this(...);
```

3.9. Zustandsänderung und Zuweisung

Bisher: Haben nie Felder eines Objekts mutiert

AUP → move() → neue Instanz von AUP

vs.

Mutation der Felder

Es gilt:

- Nicht mutierbare Objekte sind leicht(er) zu verstehen (Werte der Felder unabhängig von der Ausführungsgeschichte des Programms)
- leicht kombinierbar
- Java Bibliotheken verwenden häufig nicht mutierbare Objekte (z.B. String)

→ Klassen sollten nicht mutierbar sein, es sei denn es gibt einen *sehr guten* Grund dafür

→ Mutierbarkeit mit PS so weit wie möglich einschränken

→ Mutation ausschließlich über Methodenaufruf!

Zustandsänderung:

- Methoden, die ausschließlich den Wert eines Feldes setzen ("setter") sind in den seltensten Fällen sinnvoll.

  Meistens: Zustandsänderung komplex
- Oft: Nicht-verwandte Klassen sollen Feld lesen, aber nicht setzen dürfen
- Feld private setzen
- public-Methode, die den Wert des Feldes zurückgibt

Operatoren für Zuweisung:

`v = e`		in Scheme: `(set! v l)`		
`v+ = e`	<=>	`v = v + e`	(mit *, /, +, -)	
`v++`	<=>	`v = v + 1`	<=>	`v = v + 1`
`v--`	<=>	`v = v - 1`	<=>	`v = v - 1`

Methoden mit Seiteneffekt geben häufig keinen Wert zurück.

Rückgabetyp: *void*

3.10. Lokale Variablen

Lokale Variablen können jeweils zu Beginn eines *Blocks* eingeführt werden:

{
    t_1 v_1
    t_2 v_2
    ...
}

Innerhalb des Blocks sichtbar:

v_1, ..., v_n und alle Variablen aus umschließenden Blöcken. (Scheme: let*).

Rumpf von Methoden, Konstruktoren sind ebenfalls Blöcke.

```
class Scope {
  int x;

  void m1(int p) {
```

int x; — sichtbar in gesamter Klasse mit this

void m1(int p) { — p ist in gesamter Methode sichtbar
int y, x in der Methode sichtbar,
sequentiell gebunden
(let* ((y 0) (x (+ 42 y)))
...)

```
    int y;
    int x = 42 + y;
    ...
  }

  void m2(int p) {
    int x;
    {
      int x = 1;
      code 1;
    }
    {
      int y = 2;
      code2;
    }
  }
}
```

Insgesamt drei Sichtbarkeitsbereiche
1) gesamte Methode
2,3) zwei Blöcke auf gleicher Ebene

```
(let* ((x 0))
     (begin
          (let* ((x 1))
               code1)
          (let* ((y 2))
               code2)))
```

3.11. Schlüsselwort static

- steht vor Methoden und Feldern
- assoziiert Methode oder Feld mit der Klasse statt einer Instanz einer Klassen
- Aufruf einer static-Methode ohne Instanz einer Klasse möglich: `KlassenName.Methode(...)`
- static Methoden können nur auf static Methoden und Felder zugreifen
- static Felder: Alle Instanzen teilen sich das Feld/Wert

3.12. Schleifen

- Java: Endrekursive Aufrufe benötigen Speicher (→ SECD), Widerspruch zur Motivation der Endrekursion.
- stattdessen: um dieses Problem zu beheben: Spezielle Syntax zur Emulation von Endrekursion: Schleifen wird vom Compiler speziell übersetzt

while-Schleifen	← überführbar →	for-Schleifen
while(e_c) Sb		for(e|v; e_c; e_s) Sb
e_c: Ausdruck, der zu boolean auswertet, "Bedingung" S_b: Startwert, "Rumpf"		e|v: Variablendeklaration oder Ausdruck, "Initialisierung" e_c: Ausdruck, der zu boolean auswertet e_s: Ausdruck "(Fort)schritt" Sb: Startwert "Rumpf"

Auswertung while:
Werte e_c aus. Bei Ergebnis true, werte S_b aus und beginne erneut mit der Auswertung von while.

Auswertung for:
1) Werte e|v
2) Werte e_c aus. Ergibt sich true,
   werte S_b aus, dann e_s.
   Fahre mit Schritt 2) fort.

→ berechnete Werte müssen über Zuweisung "zurückgegeben" werden.
Schleifen haben keinen Rückgabewert!

3.13. Arrays

Häufig: Feste Zahl von Werten gleichen Typs, z.B. Vektor
Wie repräsentieren?
Wenn:

– Werte alle vom selben Typ
– Zahl der Werte fix
→ Array verwenden

Array:
– Sammlung von Variablen
– Zugriff über Index (int-Zahl)
– feste Anzahl von Elementen (Werte im Array)
– Alle Elemente haben den Typ t (oder sind Subtyp von t)

Beispiel:
1)

```
String[] wdays = {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};
```

Typ: Array von Strings (String[]) — Literal für Arrays ({"Mo", …, "So"})

Länge: 7 Einträge (automatisch berechnet wegen Initialisierung)

2)

```
int[] f = new int[10];
```

Typ: Array von int (int[]) — Aufruf Konstruktor, 10 Einträge, initialisiert mit 0) (new int[10])

Arrays gelten als Referenztypen
Defaultwert für Referenztypen ist *null*.
null ist ein Wert, der von allen anderen Werten unterschiedlich (im Sinne von ==) ist.

Zugriff auf Elemente:
Das erste Element hat immer den Index 0, das letzte (Länge – 1).

Beispiel:
`wdays[0]` → `"Mo"`
Zugriff auf Element 0.
`wdays[3]` → `"Do"`

`wdays[10]` → Fehler

`wdays.length` : Länge des Arrays

Multidimensionale Arrays

→ Arrays mit mehreren Dimensionen

Beispiel:

```
double[][] a = {{1.0, 0.0}, {0.0, 1.0}};
a[0][0] → 1.0
a[1][0] → 0.0
```

3.14. Parametrische Polymorphie (mit Java Generics)

Listenimplementierung mit Supertyp Object

– Liste konnte alle Subtypen von Object einhalten

– unterschiedliche Subtypen in einer Liste

Oft: Homogene Liste: Alle Einträge vom selben Typ

Implementierung:

Möglichkeit 1:

dynamischer Typcheck mit instanceof

Möglichkeit 2:

Typ von first als Parameter der Klasse

$\texttt{class C } \underbrace{\texttt{<}T_1\texttt{, ..., }T_n\texttt{>}}_{\text{Typparameter der Klasse}} \texttt{ \{...\}}$

– T_i können in C (fast) überall stehen wo ein Typ erwartet wird

– T_i: nur Referenztyp (auch parametrischer Referenztyp)

– T_i: kann nicht als Element-Typ eines Arrays `new` T_i`[3]` verwendet werden; kein instanceof T_i; kein Aufruf statischer Methoden

Generische Methoden

$\texttt{<}T_1\texttt{, ..., }T_n\texttt{> returntyp m (parameter...) \{...\}}$

Methode ist parametrisiert über T_i.

```
map: (A → B) list(A) → list(B)
```

3.15. Ausnahmebehandlung mit Exceptions

Programme können in Ausnahmesituationen geraten:

– Datei nicht gefunden

– Verbindung abgebrochen

Reaktionen des Programms:

– Abbruch des Programms

– Erneuter Versuch

– Benutzer benachrichtigen

→ Programme müssen diese Ausnahmesituation behandeln

Mechanismus in Java: *Exceptions*

Begriffe:

– Exception werfen: Ausnahmesituation signalisieren

– Exception fangen: Auf Ausnahmesituationen reagieren

– Exception ist Objekt, das die Ausnahmesituation näher beschreibt

Trennung werfen/fangen: Exceptions können z.B.in einer Bibliothek geworfen werden und unterschiedliche Programme möchten unterschiedlich reagieren.

→ Wiederverwendbarkeit

1) Exception werfen

Syntax: `throw e;`

Ausdruck `e` muss zu einem Objekt auswerten; Subtyp von Throwable

Exception, die Ausnahmesituation beschreibt

2) Exception fangen

Syntax:

```
try
      body
catch(E1 x1)
      catch body
      ...
```

(catch-Block: Exception-Handler für Exception E1)

Wird während der Auswertung von `body` eine Exception geworfen und gibt es einen Handler für den Typ der Exception, geht die Auswertung mit dem entsprechenden catch-body weiter. Die Exception ist in `catch body` an x1 gebunden.

Geprüfte und ungeprüfte Exceptions
1) ungeprüfte Exceptions:

- es gibt keinen Zwang einen Handler zu installieren
- signalisieren *Programmierfehler*, die nicht zur Laufzeit behebbar sind
  - Vertrag für Methode verletzt
  - Typ-Fehler, Arithmetikfehler (Division durch 0)
- werden selten gefangen: Abbruch des Programms

2) geprüfte Exceptions:

- Methode, die geprüfte Exceptions werfen können, müssen das in der Signatur dokumentieren:

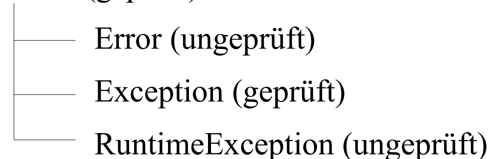
```
type m(...) throws E1, ...,En {...}
```

Exception-Typen ($E_1, \ldots, E_n$)

- Aufrufer müssen Handler installieren.
  Sonst: Fehler beim Kompilieren
- Nur für behebbare Fehler:
  - Datei nicht gefunden
  - Verbindung abgebrochen

3) Exceptions definieren:

- Standard Exception verwenden wenn möglich!
- Subklasse von: Throwable (geprüft)



(Error, Exception, RunTimeException extends Throwable)

4. Grafische Benutzeroberflächen in Java

GUI: graphical user interface

Bestandteile:

- Fenster, Knöpfe, Menüs, Mauszeiger, Scrollbalken...

Aufgaben:

- Anordnen und Anzeigen der Komponenten
- Reaktion auf Benutzerinteraktion

GUI's in Java werden mit dem Callback- oder Event-Loop-Model programmiert.

Mit einer reaktiven GUI-Komponente wird Code assoziiert, der bei deren Aktivierung ausgeführt wird. →Callback

Programm im Callback-Modell

- Das laufende Programm besteht aus der Event-Loop. Die Event-Loop ist Bestandteil des GUI-Systems und läuft endlos.
- Registriert Benutzerinteraktionen, ordnet sie einer GUI-Komponente zu und führt die Callbacks aus.

→ Unser Programm besteht hauptsächlich aus Callbacks, AWT und Swing

AWT: Abstract Windowing Toolkit

Später abgelöst von Swing-Klassen (teilweise)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
```

Anmerkung: Frame1.java läuft endlos.

Geordneter Abbruch mit `System.exit(0);`

GUI-Komponenten innerhalb eines Fensters müssen in einem *Container* plaziert werden, der bestimmt, wie die Komponenten räumlich angeordnet werden.

Häufiger Container: Panel → JPanel

Ein Panel hat eine Strategie, nach der es seine Komponenten anordnet, ein sogenanntes *Layout*.

Standardmäßig: FlowLayout

Layout:

- FlowLayout: Anordnung der Komponenten von links nach rechts oder oben nach unten, sowie es passt.
- BoxLayout: Wahlweise neben- oder übereinander
- BorderLayout: Komponente vergrößern, so dass sie in die Regionen N, S, W, O, Center passen

4.1. Reaktivität mit Callbacks

In Java sind Callbacks Methoden eines Objekts, dessen Klasse "Listener" genannt wird.

Beispiel:

Eine Listener-Klasse für JButton muss das Interface ActionListener implementieren.

Nur eine Methode:

```
public void actionPerformed(ActionEvent e)
```

Callback registrieren

```
JButton addActionListener(...)
```

Mehrere Callbacks pro Komponente möglich:

Bei Aktivierung werden sie nacheinander ausgeliefert.

Notation kürzer durch "anonyme innere Klassen"

```
button.addActionListener(new
        ActionListener() {
            public void actionPerformed(ActionEvent e) {
                ...
            }
        });
```

Erzeugt Instanz einer namenlosen Klasse, die das Interface ActionListener implementiert. Ideal wenn Callback sehr wenig Code hat.

4.2. Model-View-Controller-Architecture (MVC)

GUI-Programme werden schnell unübersichtlich.

Wünschenswert: Systematischer Aufbau aus kleinen und überschaubaren Komponenten mit Satz allgemeiner Schnitstellen

Hilfreich für GUI-Programme:

MVC-Pattern

Programm wird in 3 Bestandteile zerlegt:

- Model:
  Die Daten, mit denen sich das Programm beschäftigt
- View:
  Die Daten (werden z.B. durch GUI-Komponenten) dargestellt: Code dafür gehört zum View. Reagiert auf Veränderungen des Models.

– Controller:

Daten können durch Aktivierung der GUI-Komponente verändert werden. Aktivierung von Callbacks, gekapselt in Controllern

Entwicklung unabhängig voneinander:

Später zusammensetzen.

Vorteil: Ein Model leicht mit mehreren Views veränderbar.

4.3. Observer-Pattern

Bisherige Lösung für Counter hat einen Nachteil:

Countermodell kann nur einen View haben.

Wie lässt sich über View abstrahieren?

→ In Java eingebaut: Klasse Observable

Erlaubt einem Modell mehrere Views ("Beobachter") zu haben.

1) Modellableiten von Observable
2) Interface CounterView durch Observer ersetzen

} Observer-Pattern

Änderung im Modell:

- `setChanged()` Modell als geändert markieren
- `notifyObservers(Object o)` : Ruft die `update()`-Methoden der Observer auf, sofern `setChanged()` nach dem letzten Aufruf von `notifyObservers()` aufgerufen wurde.

  `Object o` : Nachricht ("Event"), das an die Observer weitergereicht wird.
- `addObserver(Observer o)` Observer anschließen/registrieren

Im Observer:

- Interface `Observer` implementieren:

```
void update(Observable m, Object o)
```

(m: Modell, o: Event)

5. Systematischer Umgang mit Fehlern

Testfälle helfen Fehler in Programmen frühzeitig zu erkennen. Doch: Wie findet man die Ursache im Programm? ("Debugging")

Häufig > 50% der Entwicklungszeit.

Für großer (aber auch kleine) Projekte empfiehlt sich das TRAFFIC-Prinzip

Traffic-Prinzip

1) Tracking:

Dokumentiere den Fehlern mit allen relevanten Informationen:

- Version der Software, Betriebssystem
- Wie kann der Fehler reproduziert werden
- Erwartetes Verhalten
- Beobachtetes Verhalten (Fehlermeldung)

(Große Projekte verwalten Fehlerberichte in Datenbanken)

2) Reproduce:

Versuche den Fehler zu reproduzieren, evtl. genauere Problembeschreibung verlangen. Dokumentieren.

3) Automate:

Entsprechenden Testfall erzeugen. Versuche den Testfall zu vereinfachen!

4) Find infection regions:

Welche Teile des Programms sind involviert?

Suche rückwärts vom Auftreten des Fehlers nach involvierten

Code-Sinn: Eingrenzen.

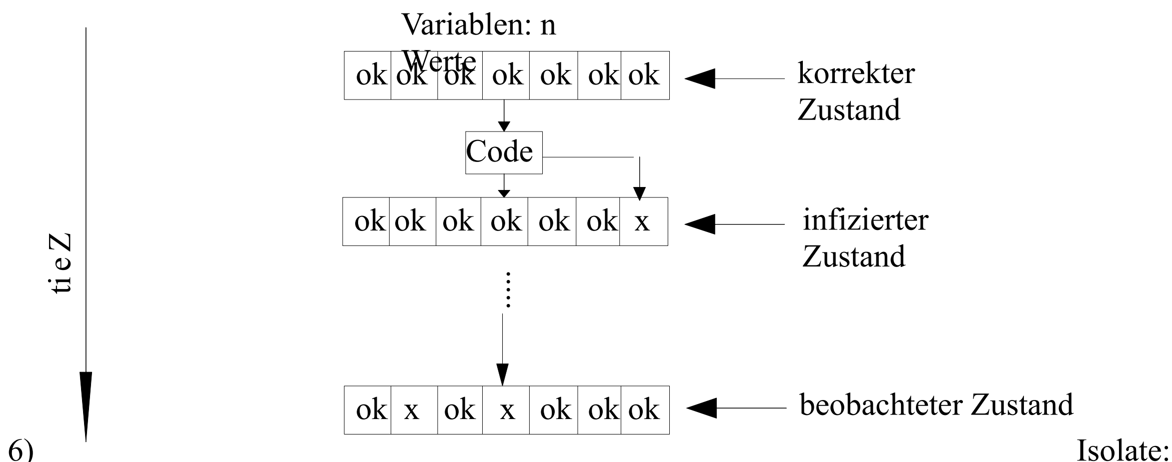
Bestimmen der "infizierten" Zustände.

} Hilfe durch Debugger

5) Focus:

Betrachte die warscheinlichen Ursachen für infizierte Zustände.

Programmablauf als Folge von Zuständen:



6) Isolate:

Lässt sich von der vermuteten Ursache eine kausale Kette von internen Zuständen konstruieren, (vorwärts durch das Programm), so dass das beobachtete Verhalten erklärbar wird?

7) Correct:

Behebe den Fehler!

Dokumentieren.

Literatur zur Fehlersuche

Andreas Zeller: "Why programs fail"

5.1. Debugger

Viele Programmiersprachen bringen einen Debugger mit.

Aufgabe:

- Unterbrechung des Programms (Breakpoints)
- Inspizieren der Werte und Laufzeit Informationen:
  - Stack
  - Lokale Variablen, Felder, Argumente
- Schrittweise (zeilenweise) Ausführung des Programms

Bei Java mitgeliefert: *jdb (Java Debugger)*

(Hohe Ähnlichkeit zu Debugger für andere Sprachen)

Der Debugger kann direkt Bezug auf den Sourcecode nehmen, wenn die `.class`-Dateien *Debug-Informationen* enthalten.

Debug-Informationen mit einkompilieren:

```
java -g <java Datei>
```

`-g` : alle Informationen einkompilieren

Start:

```
jdb <Klasse>
```

- Breakpoints
  - Stack: Informationen darüber, von wo aus eine bestimmte Methode aufgerufen wurde
  - Inspektion der lokalen Variablen, Parameter und this
  - Schrittweise Ausführung des Programms von einem bestimmten Zustand aus
- Modizifieren von Werten (`set`)

- Watchpoints
  - Finde alle Zugriffe auf ein Feld, Variable oder Array-Element. Funktionsweise wie Breakpoints. Das Programm wird jedoch nur bei Zugriff auf Feld unterbrochen.
  - Traces
  - Reihenfolge des Programmablaufs untersuchen
- Inspektion
  - `print`: externe Repräsentation des Objekts + Id für Instanz
  - `dump`: Objekt mit seinen Feldern ausdrucken
  - `eval expr`: Ausdruck im momentanen Zustand auswerten

Beispiel: Debugging eines Programms zur binären Suche

Binäre Suche:

(Schnelles) Verfahren, um festzustellen, ob bestimmter Wert in einem sortierten Array (n: Länge des Arrays) enthalten ist.

Einfachstes Verfahren:

Suche von links nach rechts im Array. Braucht max. n rekursive Aufrufe, um den Eintrag zu finden.

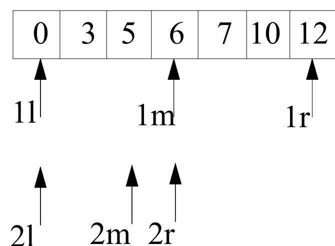
Binäre Suche nutzt die Sortierung aus:

Idee: Halbieren des Suchbereichs (rekursiv)

- Index des linken und rechten Rands des Suchbereichs
- Index der Suchbereichmitte

Beispiel:

Gesucht: 3



Lohnt sich für große Arrays: log(n) Schritte

→ kommt schneller zum Ergebnis