Vorlesung 02: Komposition und Vereinigung von Klassen

Peter Thiemann

Universität Freiburg, Germany

SS 2010

Inhalt

Komposition und Vereinigung von Klassen
- Komposition von Klassen
- Methoden für Klassenkomposition
- Vereinigung von Klassen
- Methoden für Vereinigungen von Klassen
- Rekursive Klassen
- Methoden auf rekursiven Klassen

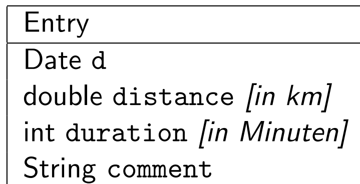
Objekte, die Objekte enthalten

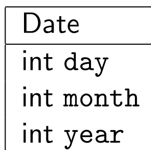
Entwickle ein Programm, das ein Lauftagebuch führt. Es enthält einen Eintrag pro Lauf. Ein Eintrag besteht aus dem Datum, der zurückgelegten Entfernung, der Dauer des Laufs und einem Kommentar zum Zustand des Läufers nach dem Lauf.

- Eintrag besteht logisch aus vier Bestandteilen
- Das Datum hat selbst Bestandteile (Tag, Monat, Jahr), deren Natur aber für das Konzept „Eintrag“ nicht wichtig sind.

Eintrag im Lauftagebuch

Klassendiagramm





Eintrag im Lauftagebuch

Implementierung

```
1  // ein Eintrag in einem Lauftagebuch
2  class Entry {
3    Date d;
4    double distance; // in km
5    int duration; // in Minuten
6    String comment;
7
8    Entry(Date d, double distance, int duration, String comment) {
9      this.d = d;
10     this.distance = distance;
11     this.duration = duration;
12     this.comment = comment;
13   }
14 }
```

Eintrag im Lauftagebuch

Beispielobjekte

- Beispieleinträge
  - am 5. Juni 2003, 8.5 km in 27 Minuten, gut
  - am 6. Juni 2003, 4.5 km in 24 Minuten, müde
  - am 23. Juni 2003, 42.2 km in 150 Minuten, erschöpft
- ... als Objekte in einem Ausdruck

```
new Entry (new Date (5,6,2003), 8.5, 27, "gut")
new Entry (new Date (6,6,2003), 4.5, 24, "müde")
new Entry (new Date (23,6,2003), 42.2, 150, "erschöpft")
```

- ... in zwei Schritten mit Hilfsdefinition

```
Date d1 = new Date (5,6,2003);
Entry e1 = new Entry (d1, 8.5, 27, "gut");
```

Eintrag im Lauftagebuch

Organisation der Beispiele in Hilfsklasse

```
1  // Beispiele für die Klasse Entry
2  class EntryExample {
3      Date d1 = new Date (5,6,2003);
4      Entry e1 = new Entry (this.d1, 8.5, 27, "gut");
5
6      Date d2 = new Date (6,6,2003);
7      Entry e2 = new Entry (this.d2, 4.5, 24, "müde");
8
9      Date d3 = new Date (23,6,2003);
10     Entry e3 = new Entry (this.d3, 42.2, 150, "erschöpft");
11
12     EntryExample () {
13     }
14 }
```

Beispiel: Zugfahrplan

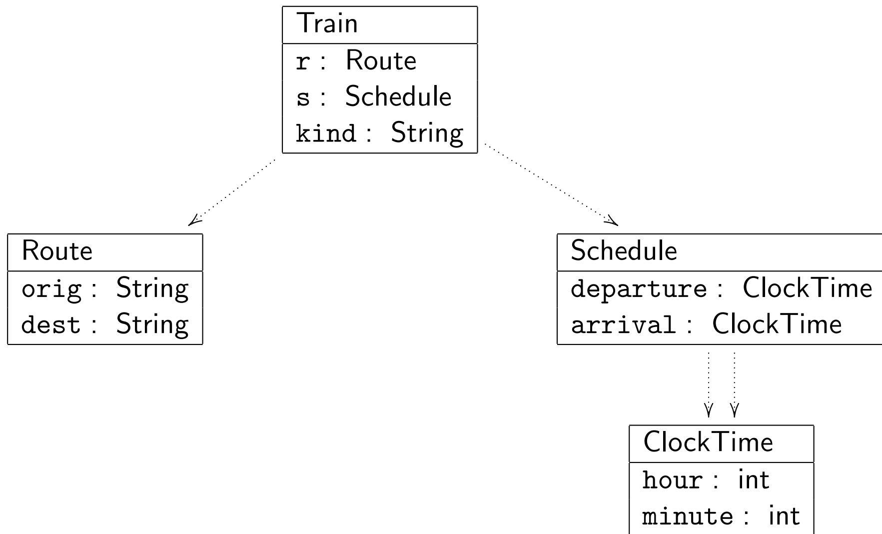
In einem Programm für Reiseauskünfte müssen Informationen über den Zugfahrplan vorgehalten werden. Für jeden Zug vermerkt der Plan die Strecke, die der Zug fährt, die Verkehrszeiten sowie die Information, was für eine Art von Zug es sich handelt (RB, RE, EC, ICE, ...). Die Strecke wird durch den Start- und den Zielbahnhof bestimmt. Eine Verkehrszeit definiert die Abfahrts- und die Ankunftszeit eines Zuges.

- Ein Zug besteht aus drei Komponenten: Strecke, Verkehrszeit, Schnellzug.
- Strecken und Verkehrszeiten bestehen aus jeweils zwei Komponenten.
- Eine Verkehrszeit enthält zwei Zeitangaben, die selbst aus Stunden und Minuten bestehen.

$\Rightarrow$ Neuigkeit: Schachtelungstiefe von Objekten > 2

Beispiel: Zugfahrplan

Klassendiagramme



Beispiel: Zugfahrplan / Implementierung

```
// eine Zugfahrt
class Train {
  Route r;
  Schedule s;
  String kind;

  Train(Route r, Schedule s, String kind) {
    this.r = r;
    this.s = s;
    this.kind = kind;
  }
}
```

```
// eine Verkehrszeit
class Schedule {
  ClockTime departure;
  ClockTime arrival;

  Schedule(ClockTime departure,
           ClockTime arrival) {
    this.departure = departure;
    this.arrival = arrival;
  }
}
```

```
// eine Bahnstrecke
class Route {
  String orig;
  String dest;

  Route(String orig, String dest) {
    this.orig = orig;
    this.dest = dest;
  }
}
```

```
// eine Uhrzeit
class ClockTime {
  int hour;
  int minute;

  ClockTime(int hour, int minute) {
    this.hour = hour;
    this.minute = minute;
  }
}
```

Beispiel: Zugfahrplan

Beispielzüge

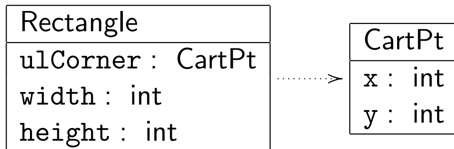
```
Route r1 = new Route ("Freiburg", "Dortmund");
Route r2 = new Route ("Basel", "Paris");

ClockTime ct1 = new ClockTime (13,04);
ClockTime ct2 = new ClockTime (18,20);
ClockTime ct3 = new ClockTime (14,57);
ClockTime ct4 = new ClockTime (18,34);

Schedule s1 = new Schedule (ct1, ct2);
Schedule s2 = new Schedule (ct3, ct4);

Train t1 = new Train (r1, s1, "ICE");
Train t2 = new Train (r2, s2, "TGV");
```

Erstellen einer zusammengesetzten Klasse

1. Identifiziere die beteiligten Klassen und erstelle Klassendiagramme. Gehe dabei top-down vor.
2. Übersetze die Klassendiagramme in Klassendefinitionen. Beginne dabei mit den einfachen Klassen, die keine Felder von Klassentyp enthalten.
   (Zusammengesetzte Klassen heißen auch *Aggregate* oder *Kompositionen*)
3. Illustriere **alle** Klassen durch Beispiele. Beginne hierbei mit den einfachen Klassen.

Methoden für Klassenkomposition

- Für ein Zeichenprogramm wird ein Rechteck durch seine linke obere Ecke sowie durch seine Breite und Höhe definiert:



- Zu einem Rechteck soll durch eine Method `distTo0()` der Abstand des Rechtecks vom Koordinatenursprung bestimmt werden.

Delegation

Gleiche Methode in übergeordneter und untergeordneter Klasse

- In Rectangle

```
// berechne den Abstand dieses Rectangle−Objekts vom Ursprung
double distTo0() { ... this.ulCorner ... this.width ... this.height ... }
```

- Der Abstand des Rechtecks `r` ist gleich dem Abstand der linken oberen Ecke `r.ulCorner` vom Ursprung.
- Umständlich in Rectangle
- Alle notwendigen Informationen liegen in CartPt.

Delegation

Gleiche Methode in übergeordneter und untergeordneter Klasse

- In Rectangle

```
// berechne den Abstand dieses Rectangle−Objekts vom Ursprung
double distTo0() { ... this.ulCorner ... this.width ... this.height ... }
```

- Der Abstand des Rechtecks `r` ist gleich dem Abstand der linken oberen Ecke `r.ulCorner` vom Ursprung.
- Umständlich in Rectangle
- Alle notwendigen Informationen liegen in CartPt.

⇒ In CartPt ist eine weitere `distTo0`-Methode erforderlich:

```
// berechne den Abstand dieses CartPt−Objekts vom Ursprung
double distTo0() { ... this.x ... this.y ... }
```

- Die Implementierung von `distTo0` in Rectangle verweist auf die Implementierung in CartPt.

Delegation

Weiterreichen von Methodenaufrufen

- In Rectangle

```
// berechne den Abstand dieses Rectangle-Objekts vom Ursprung
double distTo0() {
  return this.ulCorner.distTo0();
}
```

Die Rectangle-Klasse *delegiert* den Aufruf der `distTo0`-Methode an die CartPt-Klasse.

- In CartPt

```
// berechne den Abstand dieses CartPt-Objekts vom Ursprung
double distTo0() {
  return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

Erinnerung: Testen der distToO Methode

- Testen von CartPt

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CartPtTests {
  public static double delta = 10E-5;

  @Test
  public void testDistToOrigin() {
    assertEquals(0, new CartPt(0,0).distToOrigin(), delta);
    assertEquals(5, new CartPt(3,4).distToOrigin(), delta);
    assertEquals(10, new CartPt(-6,8).distToOrigin(), delta);
  }
```

- Testen von Rectangle . . .

Entwurf von Methoden auf Klassenkompositionen

1. Erkläre kurz den Zweck der Methode (Kommentar) und definiere die Methodensignatur. Definiere auch Methodensignaturen mit Löchern für eventuell erforderliche Hilfsmethoden auf den untergeordneten Klassen.
2. Gib Beispiele für die Verwendung der Methode.
3. Fülle den Rumpf der Methode gemäß dem Muster
   - **this** und die Felder **this**.*feldname* dürfen vorkommen
   - alle Parameter dürfen vorkommen
   - alle Methodenaufrufe auf untergeordneten Objekten dürfen vorkommen
4. Schreibe den Rumpf der Methode. Stelle fest, welche untergeordneten Methodenaufrufe erforderlich sind und lege sie auf eine Wunschliste.
5. Arbeite die Wunschliste ab.
6. Definiere die Beispiele als Tests. Teste beginned mit den innersten einfachen Objekten.

Objekte mit unterschiedlichen Ausprägungen

In einem Zeichenprogramm sollen verschiedene geometrische Figuren in einem Koordinatensystem (Einheit: ein Pixel) dargestellt werden. Zunächst geht es um drei Arten von Figuren:

- *Quadrate mit Referenzpunkt links oben und gegebener Seitenlänge,*
- *Kreise mit dem Mittelpunkt als Referenzpunkt und gegebenem Radius und*
- *Punkte, die nur durch den Referenzpunkt gegeben sind und als Scheibe mit einem Radius von 3 Pixeln gezeichnet werden.*

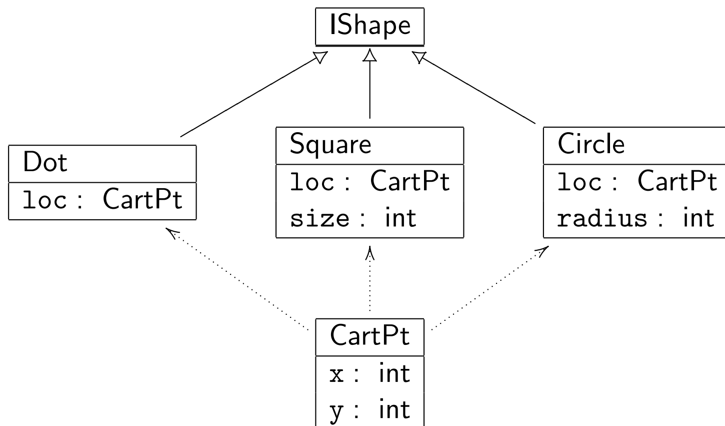
Vereinigung von Klassen

Klar Jede Art Figur kann durch eine zusammengesetzte Klasse repräsentiert werden. Der Referenzpunkt wird jeweils durch ein separates Punktobjekt dargestellt.

$\Rightarrow$ drei unterschiedliche Klassen, deren Objekte nicht miteinander verträglich sind

Gesucht Ein Typ IShape, der Objekte aller Figurenklassen umfasst. D.h., die *Vereinigung* der Klassentypen.

Figuren im Klassendiagramm



Interface und Implementierung

- Die Klassentypen Dot, Square, Circle werden zu einem gemeinsamen *Interfacetyp* zusammengefasst, angedeutet durch den offenen *Generalisierungspfeil* im Diagramm.
- Er wird durch eine *Interfacedefinition* angegeben:

```
1 // geometrische Figuren
2 interface IShape { }
```

- Die Klassendefinition gibt an, ob eine Klasse zu einem Interface gehört oder nicht. Dies geschieht durch eine *implements-Klausel*.

```
1 // ein Punkt
2 class Dot implements IShape {
3   CartPt loc;
4
5   Dot(CartPt loc) {
6     this.loc = loc;
7   }
8 }
```

Weitere Implementierungen

- Ein Interface kann beliebig viele implementierende Klassen haben.

```
2  // ein Quadrat
3  class Square implements IShape {
4    CartPt loc;
5    int size;
12 }
```

```
2  // ein Kreis
3  class Circle implements IShape {
4    CartPt loc;
5    int radius;
12 }
```

Verwendung

- Square, Circle und Dot Objekte besitzen jeweils ihren Klassentyp.

```
CartPt p0 = new CartPt (0,0);
CartPt p1 = new CartPt (50,50);
CartPt p2 = new CartPt (80,80);

Square s = new Square (p0, 50);
Circle c = new Circle (p1, 30);
Dot d = new Dot (p2);
```

- Durch „`implements`“ besitzen sie **zusätzlich** den Typ IShape.

```
IShape sh1 = new Square (p0, 50);
IShape sh2 = new Circle (p1, 30);
IShape sh3 = new Dot (p2);

IShape sh4 = s;
IShape sh5 = c;
IShape sh6 = d;
```

Typfehler

- Eine Zuweisung

$$\text{Ty var} = \textbf{new } \text{Cls}(\dots)$$

  ist *typkorrekt*, falls Cls ein *Subtyp* von Ty ist. Das heißt:
  - Ty ist identisch zu Cls oder
  - Cls ist definiert mit “Cls **implements** Ty”

  Ty heißt dann auch *Supertyp* von Cls.
  Anderenfalls liegt ein *Typfehler* vor, den Java zurückweist.

- Typkorrekte Zuweisungen

```
Square good1 = new Square (p0, 50);
IShape good2 = new Square (p1, 30);
```

- Zuweisungen mit Typfehlern

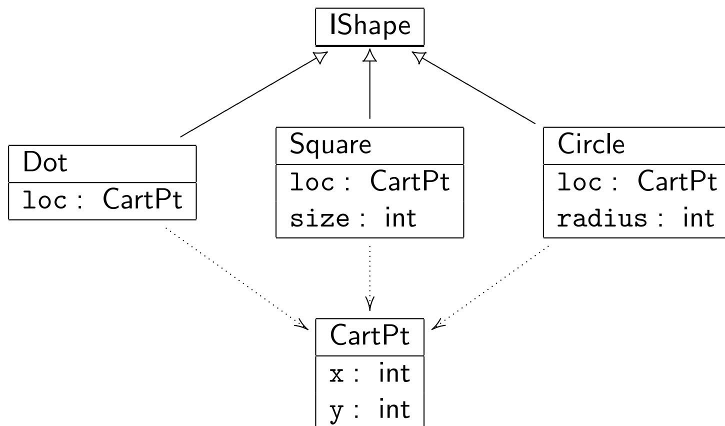
```
Square bad1 = new Circle (p0, 50);
IShape bad2 = new CartPt (20, 30);
```

Erstellen einer Vereinigung von Klassen

1. Wenn ein Datenbereich auftritt, in dem Objekte mit unterschiedlichen Attributen auftreten, so ist das ein Indiz, dass eine Vereinigung von Klassen vorliegt.
2. Erstelle zunächst das Klassendiagramm. Richte das Augenmerk zunächst auf den Entwurf der Vereinigung und verfeinere zusammengesetzte Klassen später.
3. Übersetze das Klassendiagramm in Code. Aus dem Interfacekasten wird ein Interface; die darunterliegenden Klassenkästen werden Klassen, die jeweils das Interface implementieren.
   Versehe jede Klasse mit einer kurzen Erklärung.

Methoden auf Vereinigungen von Klassen

Erinnerung: die Klassenhierarchie zu IShape mit Subtypen Dot, Square und Circle



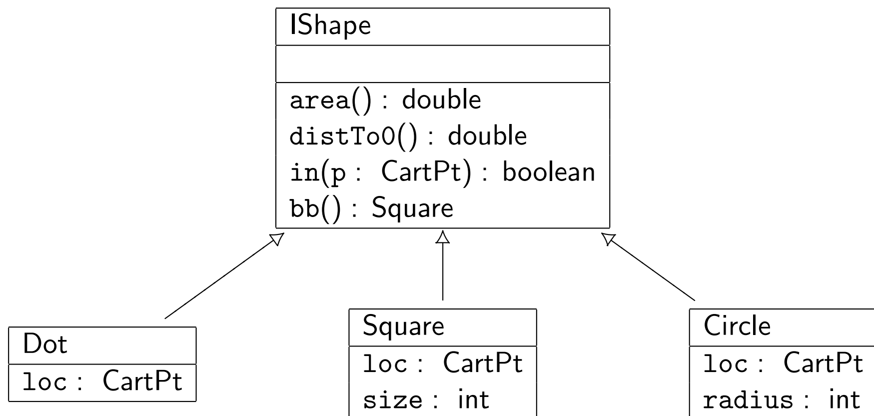
Methoden für IShape

Das Programm zur Verarbeitung von geometrischen Figuren benötigt Methoden zur Lösung folgender Probleme.

1. *double area()*
   Wie groß ist die Fläche einer Figur?
2. *double distTo0()*
   Wie groß ist der Abstand einer Figur zum Koordinatenursprung?
3. *boolean in(CartPt p)*
   Liegt ein Punkt innerhalb einer Figur?
4. *Square bb()*
   Was ist die Umrandung einer Figur? Die Umrandung ist das kleinste Rechteck, das die Figur vollständig überdeckt. (Für die betrachteten Figuren ist es immer ein Quadrat.)

Methodensignaturen im Interface IShape

- Die Methodensignaturen werden im Interface IShape definiert.
- Das stellt sicher, dass jedes Objekt vom Typ IShape die Methoden implementieren muss.



Implementierung von IShape

```
// geometrische Figuren
interface IShape {
    // berechne die Fläche dieser Figur
    double area ();
    // berechne den Abstand dieser Figur zum Ursprung
    double distTo0();
    // ist der Punkt innerhalb dieser Figur?
    boolean in (CartPt p);
    // berechne die Umrandung dieser Figur
    Square bb();
}
```

Methode `area()` in den implementierenden Klassen

- Die Definition einer Methodensignatur für Methode *m* im Interface **erzwingt** die Implementierung von *m* mit dieser Signatur in **allen** implementierenden Klassen.

⇒ `area()` in Dot:

```
// berechne die Fläche dieser Figur
double area() { ... this.loc ... }
```

⇒ `area()` in Square:

```
// berechne die Fläche dieser Figur
double area() { ... this.loc ... this.size ... }
```

⇒ `area()` in Circle:

```
// berechne die Fläche dieser Figur
double area() { ... this.loc ... this.radius ... }
```

Klasse mit Anwendungsbeispielen und Tests

```
public class IShapeTest {

  IShape dot = new Dot (new CartPt (4,3));
  IShape squ = new Square (new CartPt (4,3), 3);
  IShape cir = new Circle (new CartPt (12,5), 2);

  @Test
  public void testArea() {
    assertEquals(0.0, dot.area(), 0.1);
    assertEquals(9.0, squ.area(), 0.1);
    assertEquals(12.56, cir.area(), 0.01);
  }
}
```

Implementierungen von area()

⇒ area() in Dot:

```
double area() {
  return 0;
}
```

⇒ area() in Square:

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

⇒ area() in Circle:

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

- eine Hilfsmethode in CartPt ist nicht erforderlich

Methode `distTo0()` in den implementierenden Klassen

⇒ in Dot:

```
double distTo0() { ... this.loc ... }
```

⇒ in Square:

```
double distTo0() { ... this.loc ... this.size ... }
```

⇒ in Circle:

```
double distTo0() { ... this.loc ... this.radius ... }
```

⇒ Hilfsmethode in CartPt

```
ttt mmm() { ... this.x ... this.y ...}
```

Anwendungsbeispiele und Tests für `distTo0()`

```
public class IShapeTest {

  IShape dot = new Dot (new CartPt (4,3));
  IShape squ = new Square (new CartPt (4,3), 3);
  IShape cir = new Circle (new CartPt (12,5), 2);

  @Test
  public void testDistToOrigin() {
    assertEquals(5.0, dot.distToOrigin(), 0.01);
    assertEquals(5.0, squ.distToOrigin(), 0.01);
    assertEquals(11.0, cir.distToOrigin(), 0.01);
  }
}
```

Analyse von `distTo0()`

- Der Abstand eines Dot zum Ursprung ist der Abstand seines `loc` Feldes zum Ursprung.
- Der Abstand eines Square zum Ursprung ist der Abstand seines Referenzpunktes zum Ursprung.
- Der Abstand eines Circle zum Ursprung ist der Abstand seines Mittelpunktes abzüglich des Radius. Falls der Kreis den Ursprung enthält, so ist der Abstand 0.

Analyse von `distTo0()`

- Der Abstand eines Dot zum Ursprung ist der Abstand seines `loc` Feldes zum Ursprung.
- Der Abstand eines Square zum Ursprung ist der Abstand seines Referenzpunktes zum Ursprung.
- Der Abstand eines Circle zum Ursprung ist der Abstand seines Mittelpunktes abzüglich des Radius. Falls der Kreis den Ursprung enthält, so ist der Abstand 0.

⇒ Die Hilfsmethode auf CartPt muss selbst den Abstand zum Ursprung berechnen:

⇒ Hilfsmethode in CartPt

```
double distTo0() {
  return Math.sqrt(this.x * this.x + this.y * this.y);
}
```

Implementierungen von distTo0()

⇒ distTo0() in Dot:

```
double double() {
  return this.loc.distTo0();
}
```

⇒ distTo0() in Square:

```
double distTo0() {
  return this.loc.distTo0;
}
```

⇒ distTo0() in Circle:

```
double distTo0() {
  return Math.max(this.loc.distTo0() − this.radius, 0);
}
```

- eine Hilfsmethode in CartPt ist nicht erforderlich

Methode `bb()` in den implementierenden Klassen

⇒ in Dot:

```
Square bb() { ... this.loc ... }
```

⇒ in Square:

```
Square bb() { ... this.loc ... this.size ... }
```

⇒ in Circle:

```
Square bb() { ... this.loc ... this.radius ... }
```

⇒ Hilfsmethode in CartPt

```
ttt mmm() { ... this.x ... this.y ...}
```

Anwendungsbeispiele und Tests für bb()

```
public class IShapeTest {

  IShape dot = new Dot (new CartPt (4,3));
  IShape squ = new Square (new CartPt (4,3), 3);
  IShape cir = new Circle (new CartPt (12,5), 2);

  @Test
  public void testBb() {
    assertEquals(new Square(new CartPt(4,3), 1), dot.bb());
    assertEquals(squ, squ.bb());
    assertEquals(new Square(new CartPt(10,3), 4), cir.bb());
  }
}
```

Einzige Schwierigkeit

Implementierung für Circle, wo ein Quadrat konstruiert werden muss, das um eine Radiusbreite vom Mittelpunkt des Kreises entfernt ist.

Implementierungen von bb()

⇒ bb() in Dot:

```
Square bb() {
  return new Square(this.loc, 1);
}
```

⇒ bb() in Square:

```
Square bb() {
  return this;
}
```

⇒ bb() in Circle:

```
Square bb() {
  return new Square(this.loc.translate(−this.radius), 2∗this.radius);
}
```

- Wunschliste: Hilfsmethode `translate (offset`: int) in CartPt, die einen um `offset` verschobenen Punkt erzeugt.

Hilfsmethode `translate` in CartPt

- Wunschliste: Hilfsmethode `translate` (`offset`: int) in CartPt, die einen um `offset` verschobenen Punkt erzeugt.

```
// Cartesische Koordinaten auf dem Bildschirm
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  CartPt translate(int offset) {
    return new CartPt (this.x + offset, this.y + offset);
  }
}
```

Alternative Implementierung für Dot

Für die Klasse Dot ist die bisherige Methode `bb()` interpretationsbedürftig:

- Ein Quadrat mit Seitenlänge 1 ist zu groß.
- Ein Quadrat mit Seitenlänge 0 ist kein Quadrat.

Je nach Anwendung kann es besser sein, einen Fehler zu signalisieren. Dafür besitzt Java *Exceptions* (Ausnahmen), die über die Methode `InputOutput.error(String message)` ausgelöst werden können.

```
Square bb() {
  return InputOutput.error ("bounding box for a dot");
}
```

Entwurf von Methoden auf Vereinigungen von Klassen

1. Erkläre den Zweck der Methode (Kommentar) und definiere die Methodensignatur. Füge die Methodensignatur jeder implementierenden Klasse hinzu.
2. Gib Beispiele für die Verwendung der Methode in jeder Variante.
3. Fülle den Rumpf der Methode gemäß dem (bekannten) Muster
   - **this** und die Felder **this**.*feldname* dürfen vorkommen
   - alle Parameter dürfen vorkommen
   - alle Methodenaufrufe auf untergeordneten Objekten dürfen vorkommen
4. Schreibe den Rumpf der Methode in jeder Variante.
5. Definiere die Beispiele als Tests.

Rekursive Klassen

Erstelle ein Programm, das ein Lauftagebuch führt. Der Läufer erstellt jeden Tag einen Eintrag, der den Lauf des Tages dokumentiert. Ein Eintrag besteht aus dem Datum, der zurückgelegten Entfernung, der Dauer des Laufs und einem Kommentar zum Zustand des Läufers nach dem Lauf.

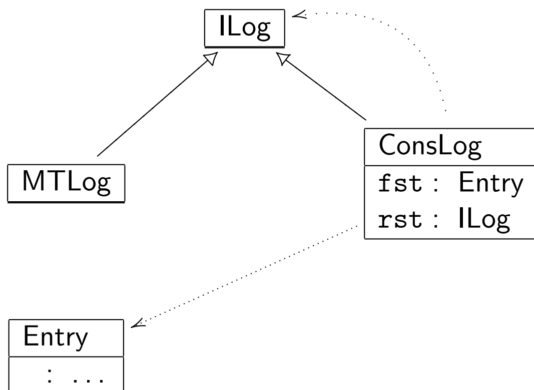
- Bereits erledigt: Klasse Entry für einzelne Einträge
- Noch zu tun: Ein Tagebuch enthält eine **beliebige Anzahl** von Entry-Objekten

$\Rightarrow$ Wunsch: repräsentiere das Tagebuch durch eine **Liste** von Einträgen

Entwurf einer Liste von Entry

- Eine Liste von Einträgen ist entweder
  - leer **oder**
  - besteht aus einem Eintrag und einer restlichen Liste von Einträgen
- Das Wort "**oder**" weist auf eine Vereinigung von Klassen hin
- Repräsentiere also eine Liste von Einträgen durch ein Interface ILog, das als Vereinigung zweier Klassen dient, die je für die leere bzw nicht-leere Liste stehen.
  - leer → Klasse MTLog
  - nicht-leer → Klasse ConsLog

Klassendiagramm zur Liste von Entry



Implementierung von ILog

```
1 // Lauftagebuch
2 interface ILog {}
```

```
1 // leeres Tagebuch
2 class MTLog implements ILog {
3   MTLog() {}
4 }
```

```
1 // Listenglied im Lauftagebuch
2 class ConsLog implements ILog {
3   Entry fst;
4   ILog rst;
5
6   ConsLog(Entry fst, ILog rst) {
7     this.fst = fst;
8     this.rst = rst;
9   }
10 }
```

Beispiel: ein Tagebuch

- Beispieltagebuch
  - am 5. Juni 2003, 8.5 km in 27 Minuten, gut
  - am 6. Juni 2003, 4.5 km in 24 Minuten, müde
  - am 23. Juni 2003, 42.2 km in 150 Minuten, erschöpft
- ... zunächst die einzelnen Einträge als Objekte

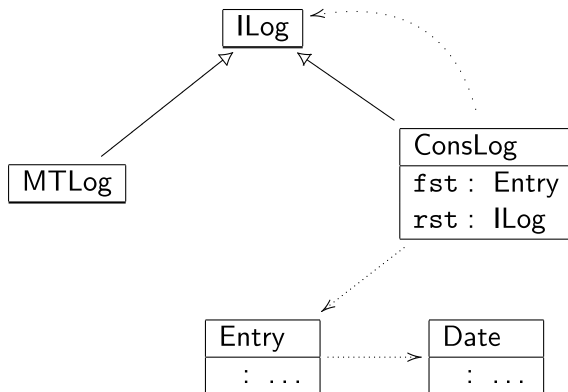
```
Entry e1 = new Entry (new Date (5,6,2003), 8.5, 27, "gut");
Entry e2 = new Entry (new Date (6,6,2003), 4.5, 24, "müde");
Entry e3 = new Entry (new Date (23,6,2003), 42.2, 150, "erschöpft");
```

- ... Aufbau der Liste. Ergebnis in `i4`.

```
ILog i1 = new MTLog ();
ILog i2 = new ConsLog (e1, i1);
ILog i3 = new ConsLog (e2, i2);
ILog i4 = new ConsLog (e3, i3);
```

Methoden auf rekursiven Klassen

Erinnerung: das Lauftagebuch



- Ziel: Definiere Methoden auf ILog

Muster: Methoden für ILog

- gewünschte Methodensignatur in ILog

```
// Zweck der Methode
ttt mmm();
```

- Implementierungsschablone in MTLog (**implements** ILog)

```
ttt mmm() { ... }
```

- Implementierungsschablone in ConsLog (**implements** ILog)

```
ttt mmm() {
  ... this.fst.nnn() ...
  ... this.rst.mmm() ... // rekursiver Aufruf
}
```

- ggf. Hilfsmethode in Entry

```
uuu nnn() { ... this.d.lll() ... this.distance ... }
```

- ggf. Hilfsmethode in Date

```
vvv lll() { ... this.day ... }
```

Beispiel: Gesamtstrecke

Ermittle aus dem Lauftagebuch die insgesamt gelaufenen Kilometer.

- in ILog

```
// berechne die Gesamtkilometerzahl
double totalDistance();
```

- Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gesamtstrecke

Implementierungen

- Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.

- Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gesamtstrecke

Implementierungen

- Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- in MTLog

```
double totalDistance() {
  return 0;
}
```

- Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.

Beispiel: Gesamtstrecke

Implementierungen

- Die Gesamtkilometerzahl für ein leeres Tagebuch ist 0.
- in MTLog

```
double totalDistance() {
  return 0;
}
```

- Die Gesamtkilometerzahl für ein nicht-leeres Tagebuch ist die gelaufene Distanz plus die Gesamtkilometerzahl des restlichen Tagebuchs.
- in ConsLog

```
double totalDistance() {
  return this.fst.distance + this.rst.totalDistance();
}
```

Beispiel: Gesamtstrecke

Bemerkung

- In Entry ist keine spezielle Hilfsmethode erforderlich. Der Zugriff auf das `distance` Feld kann auch aus der ConsLog-Methode heraus erfolgen.
- In Date ist keine spezielle Hilfsmethode erforderlich. Das Datum spielt für die Funktion keine Rolle.

Zusammenfassung

Arrangements von Mustern und Klassen

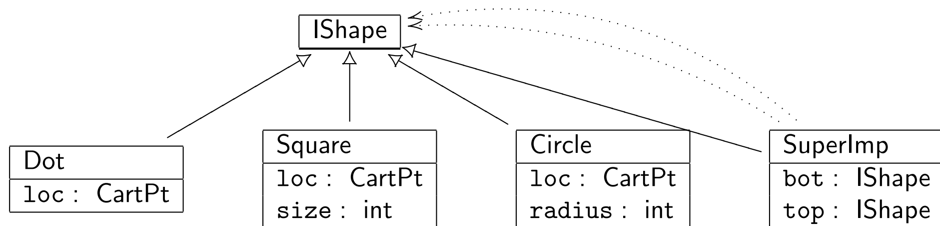
- Einfache Klassen:
  einfache Methoden, die nur die Felder der eigenen Klasse verwenden
- Zusammengesetzte Klassen:
  Methoden verwenden die eigenen Felder, sowie Methoden und Felder der enthaltenen Objekte
- Vereinigung von Klassen:
  Methoden im Interface müssen in jeder Variante definiert werden
- Rekursive Klassen:
  Beim Entwurf der Methoden wird angenommen, dass die (rekursiven) Methodenaufrufe auf dem Start-Interface bereits das richtige Ergebnis liefern.

Weiteres Beispiel: Allgemeine Baumstruktur

Ein Zeichenprogramm kennt mindestens drei Arten von Figuren: Punkte, Quadrate und Kreise. Darüber hinaus kann es auch mit Kombinationen von Figuren arbeiten: aus je zwei Figuren kann durch Übereinanderlegen eine neue Figur erzeugt werden.

- Neue Alternative für Figuren wird durch neue Implementierungsklasse von IShape definiert.
- Die neue Klasse heißt SuperImp für superimposition (Überlagerung).

Erweiterte Figuren im Klassendiagramm



Implementierung

```
1  // Überlagerung zweier Figuren
2  class SuperImp implements IShape {
3    IShape bot;
4    IShape top;
5
6    SuperImp(IShape bot, IShape top) {
7      this.bot = bot;
8      this.top = top;
9    }
10 }
```

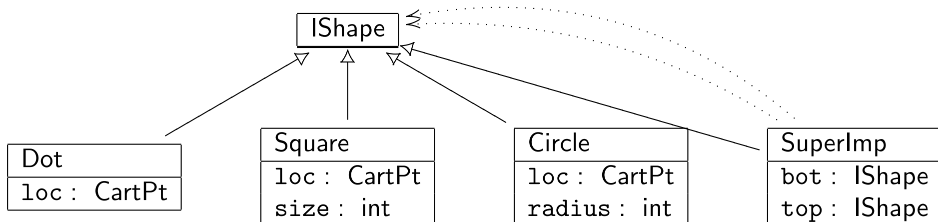
Einige Beispiele

```
1   CartPt cp1 = new CartPt (100,200);
2   CartPt cp2 = new CartPt (20, 50);
3   CartPt cp3 = new CartPt (0,0);
4
5   Square s1 = new Square (cp1, 40);
6   IShape s2 = new Square (cp2, 30);
7   Circle c1 = new Circle (cp3, 20);
8
9   IShape sh1 = new SuperImp (c1, s1);
10  IShape sh2 = new SuperImp (s2, new Square (cp1, 300));
11  IShape sh3 = new SuperImp (s1, sh2);
```

- Z6 weist ein Square-Objekt einer IShape-Variable zu
- Z9 **übergibt Objekte von einem Subtyp anstelle des erwarteten Typs**. Der Konstruktor von SuperImp erwartet Argumente von Typ IShape und erhält stattdessen Objekte der Subtypen Square bzw. Circle.
- Z10, zweites Argument, gleiches Phänomen
- **alle Zuweisungen sind typkorrekt!**

Erweiterung von IShape um Überlappung

Erinnerung: rekursive Erweiterung von IShape



- SuperImp steht für die Überlappung/Vereinigung zweier Figuren
- Ziel: Definition der IShape-Methoden `distTo0()` und `bb()` auf SuperImp

Entwicklung von `distTo0()` in SuperImp

- Das Muster

```
double distTo0() {
  ... this.bot.distTo0() ...
  ... this.top.distTo0() ...
}
```

Entwicklung von `distTo0()` in SuperImp

- Das Muster

```
double distTo0() {
  ... this.bot.distTo0() ...
  ... this.top.distTo0() ...
}
```

- Offenbar ist der Abstand der Vereinigung zweier Figuren gleich dem Minimum der Abstände. Verwende also `Math.min()`

Entwicklung von `distTo0()` in SuperImp

- Das Muster

```
double distTo0() {
  ... this.bot.distTo0() ...
  ... this.top.distTo0() ...
}
```

- Offenbar ist der Abstand der Vereinigung zweier Figuren gleich dem Minimum der Abstände. Verwende also `Math.min()`
- Ausgefülltes Muster

```
double distTo0() {
  return Math.min(this.bot.distTo0(),
                  this.top.distTo0());
}
```

Analyse von `bb()` in SuperImp

- Bisherige Methodensignatur: Square `bb()`
- Nicht adäquat für Vereinigung, da hierbei (beliebige) Rechtecke entstehen können, nicht notwendigerweise vom Typ IShape.
- Ausweg: Postuliere spezielle Klasse BoundingBox für diese Rechtecke und verschiebe ihre Definition auf später
- Revidierte Methodensignatur (in IShape)

```
// berechne die Umrandung einer Figur
BoundingBox bb();
```

Muster von bb() in SuperImp

```
BoundingBox bb() {
  // berechne die Umrandung für top
  ... this.top.bb() ...
  // berechne die Umrandung für bot
  ... this.bot.bb() ...
}
```

Diese beiden Umrandungen müssen kombiniert werden.

Muster von bb() in SuperImp

```
BoundingBox bb() {
  // berechne die Umrandung für top
  ... this.top.bb() ...
  // berechne die Umrandung für bot
  ... this.bot.bb() ...
}
```

Diese beiden Umrandungen müssen kombiniert werden.

Anforderung an BoundingBox

```
  // kombiniere diese Umrandung mit der Argument−Umrandung
  BoundingBox combine (BoundingBox that);
```

Muster von bb() in SuperImp

```
BoundingBox bb() {
  // berechne die Umrandung für top
  ... this.top.bb() ...
  // berechne die Umrandung für bot
  ... this.bot.bb() ...
}
```

Diese beiden Umrandungen müssen kombiniert werden.

Anforderung an BoundingBox

```
  // kombiniere diese Umrandung mit der Argument−Umrandung
  BoundingBox combine (BoundingBox that);
```

Ausgefülltes Muster in SuperImp

```
BoundingBox bb() {
  return this.top.bb().combine(this.bot.bb());
}
```

Implementierung von BoundingBox

```
1  // Umrandungen von 2D−Figuren
2  class BoundingBox {
3      int lft;
4      int rgt; // lft <= rgt
5      int top;
6      int bot; // top <= bot
7
8      // kombiniere diese Umrandung mit der Argument−Umrandung
9      BoundingBox combine (BoundingBox that) {
10         return new BoundingBox
11             (Math.min (this.lft, that.lft),
12              Math.max (this.rgt, that.rgt),
13              Math.min (this.top, that.top),
14              math.max (this.bot, that.bot));
15     }
16
17     BoundingBox (int lft, int rgt, int top, int bot) {} //weggelassen
18 }
```

Restliche Implementierung

- Die Implementierungen von `bb()` für Circle und Square sind jetzt einfach.
- Selbst.