Programmieren in Java

Vorlesung 02: Zusammengesetzte Klassen

Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2013

Inhalt

Fallstudie

- Aufräumen der `Street` Klasse
- `Player` Klasse, Modellierung des Besitzers
- Interfaces `IProperty`, `IAction`, `IActionCard`, `IDice`
- `Collection`, Repräsentation einer Assoziation
- `Game` Klasse
- `IField` Interface

Kollaboration

- Checkout von `https://github.com/peterthiemann/monopoly.git`
- Implementieren . . . (z.B. eine Implementierung von `IDice`)
- Paste nach `http://pastebin.com/`
- Senden des Paste an Twitter `@ProglangUniFR`
- oder `#ufrjava`

Interface IDice

```
1  package monopoly;
2
3  public interface IDice {
4    /**
5     * roll the dice
6     */
7    public void roll();
8    /**
9     * obtain the current value of the dice
10    * @return current value of the dice
11    */
12   public int getValue();
13   /**
14    * check if the roll is a double
15    * @return true if doubles have been rolled
16    */
17   public boolean isDoubles();
18 }
```

Zusammengesetzte Klassen

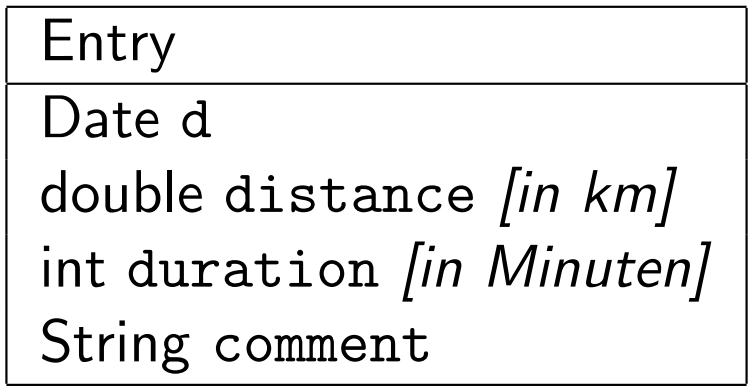
Objekte, die Objekte enthalten

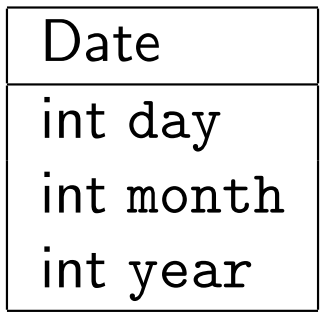
Entwickle ein Programm, das ein Lauftagebuch führt. Es enthält einen Eintrag pro Lauf. Ein Eintrag besteht aus dem Datum, der zurückgelegten Entfernung, der Dauer des Laufs und einem Kommentar zum Zustand des Läufers nach dem Lauf.

- 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

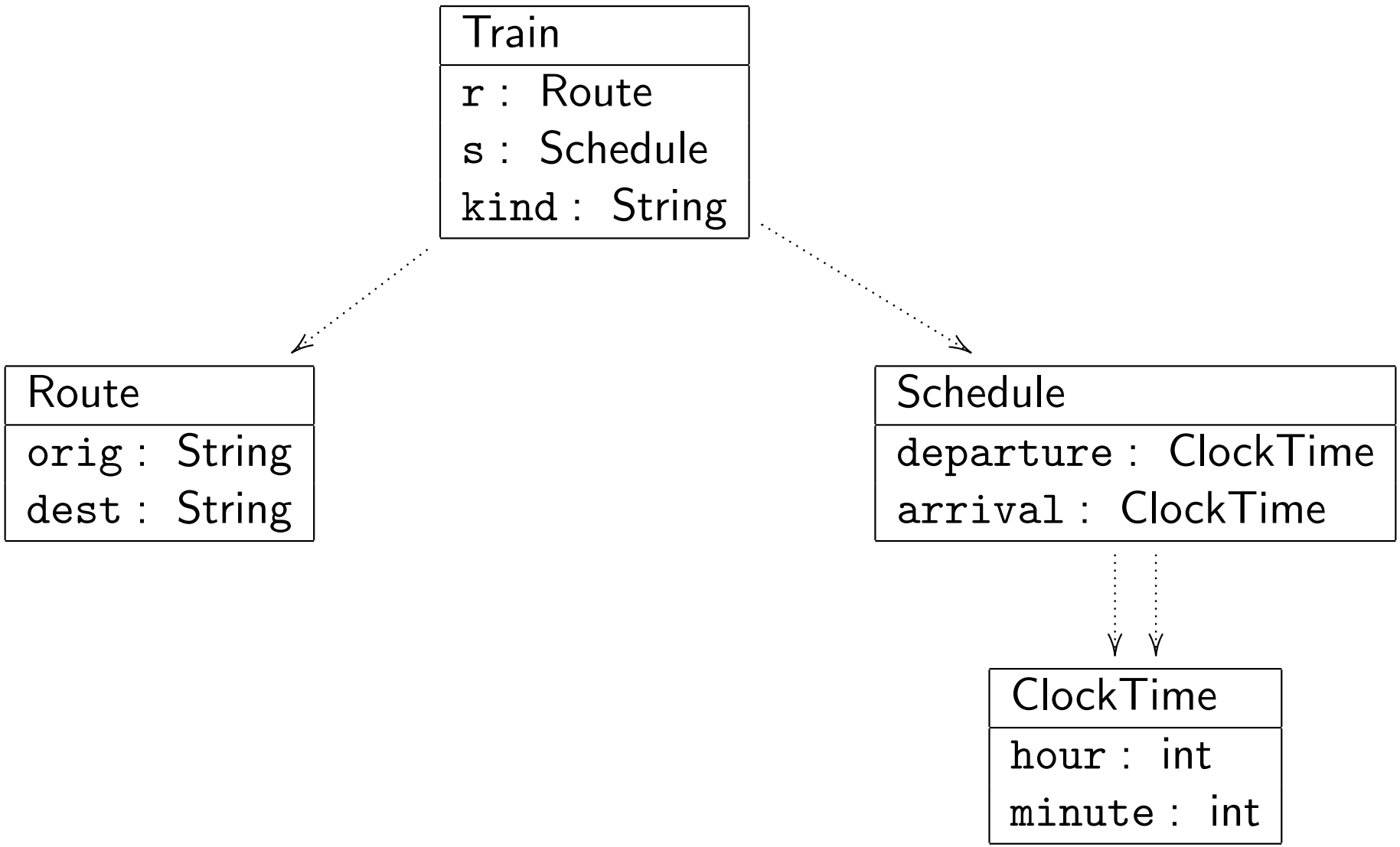
In einem Programm für Reiseauskünfte müssen Informationen über den Zugfahrplan vorgehalten werden. Für jeden Zug vermerkt der Plan die Strecke, die der Zug fährt, die Verkehrszeiten sowie die Information, was für eine Art von Zug es sich handelt (RB, RE, EC, ICE, ...). Die Strecke wird durch den Start-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

```
1  // eine Zugfahrt
2  class Train {
3    Route r;
4    Schedule s;
5    String kind;
6
7    Train(Route r, Schedule s, String kind) {
8      this.r = r;
9      this.s = s;
10     this.kind = kind;
11   }
12 }
```

```
1  // eine Verkehrszeit
2  class Schedule {
3    ClockTime departure;
4    ClockTime arrival;
5
6    Schedule(ClockTime departure,
7             ClockTime arrival) {
8      this.departure = departure;
9      this.arrival = arrival;
10   }
11 }
```

```
1  // eine Bahnstrecke
2  class Route {
3    String orig;
4    String dest;
5
6    Route(String orig, String dest) {
7      this.orig = orig;
8      this.dest = dest;
9    }
10 }
```

```
1  // eine Uhrzeit
2  class ClockTime {
3    int hour;
4    int minute;
5
6    ClockTime(int hour, int minute) {
7      this.hour = hour;
8      this.minute = minute;
9    }
10 }
```

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");
```

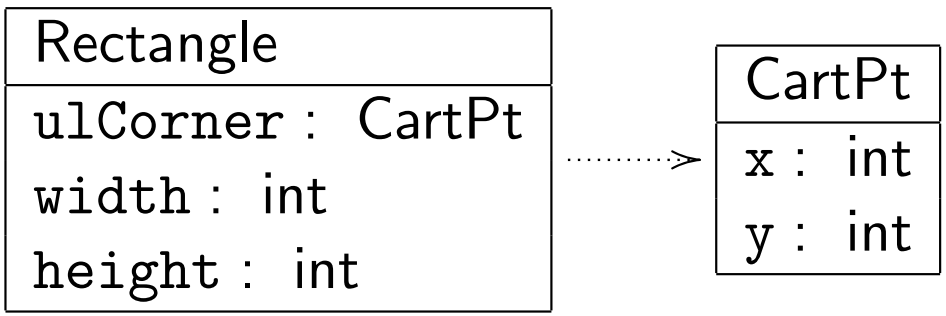
Erstellen einer zusammengesetzten Klasse

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 zusammengesetzte Klassen

Methoden für zusammengesetzte Klassen

- 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 notwendige Information liegt schon 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 notwendige Information liegt schon in CartPt.

$\Rightarrow$ 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);
}
```

Muster zur Delegation

Entwurf von Methoden auf zusammengesetzten Objekten

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.

Vereinigung von Klassen

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 wiedergegeben werden.*

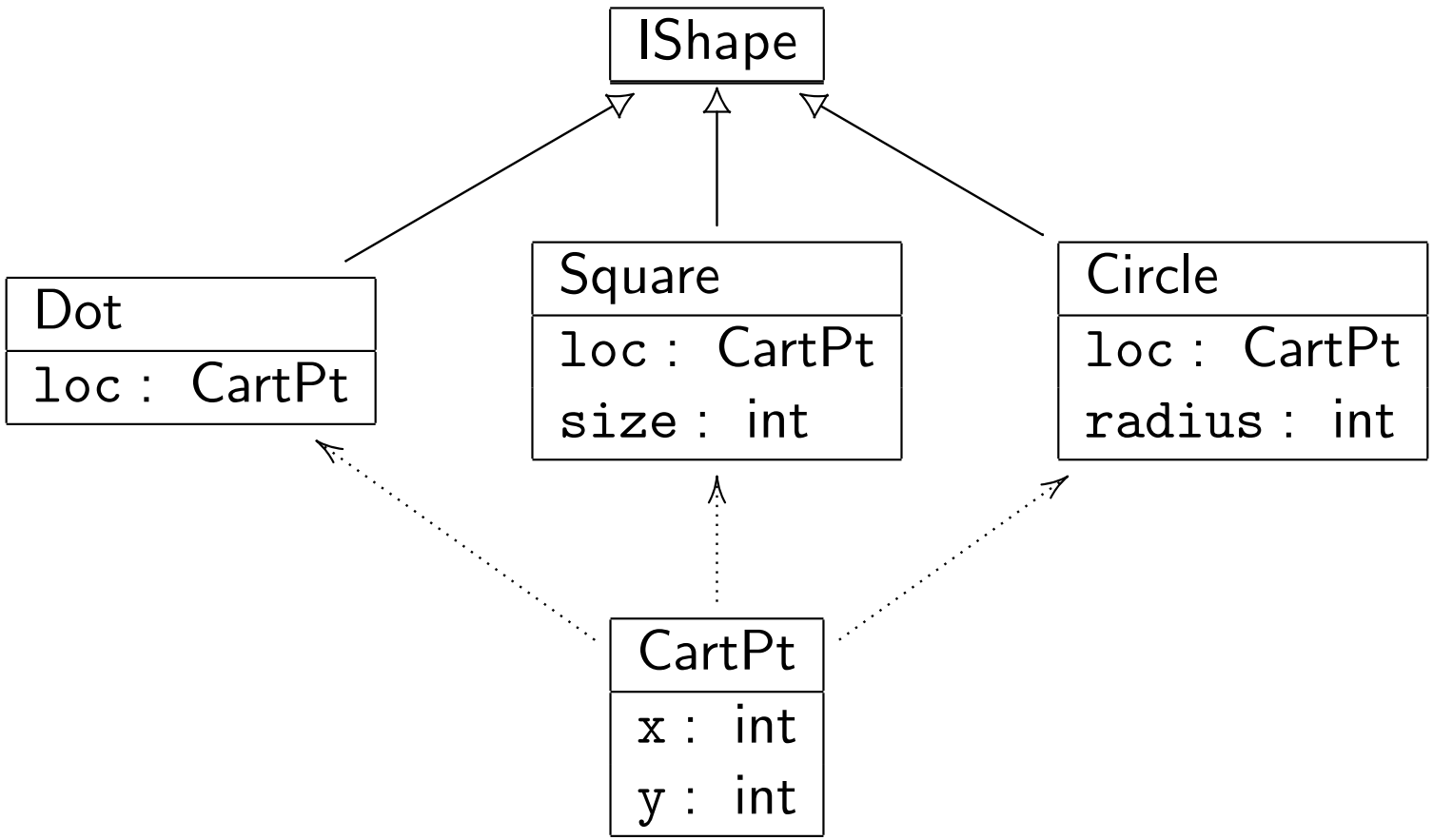
Vereinigung von Klassen

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.

```
1  // ein Quadrat
2  class Square implements IShape {
3     CartPt loc;
4     int size;
11 }
```

```
1  // ein Kreis
2  class Circle implements IShape {
3     CartPt loc;
4     int radius;
11 }
```

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

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

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);
```

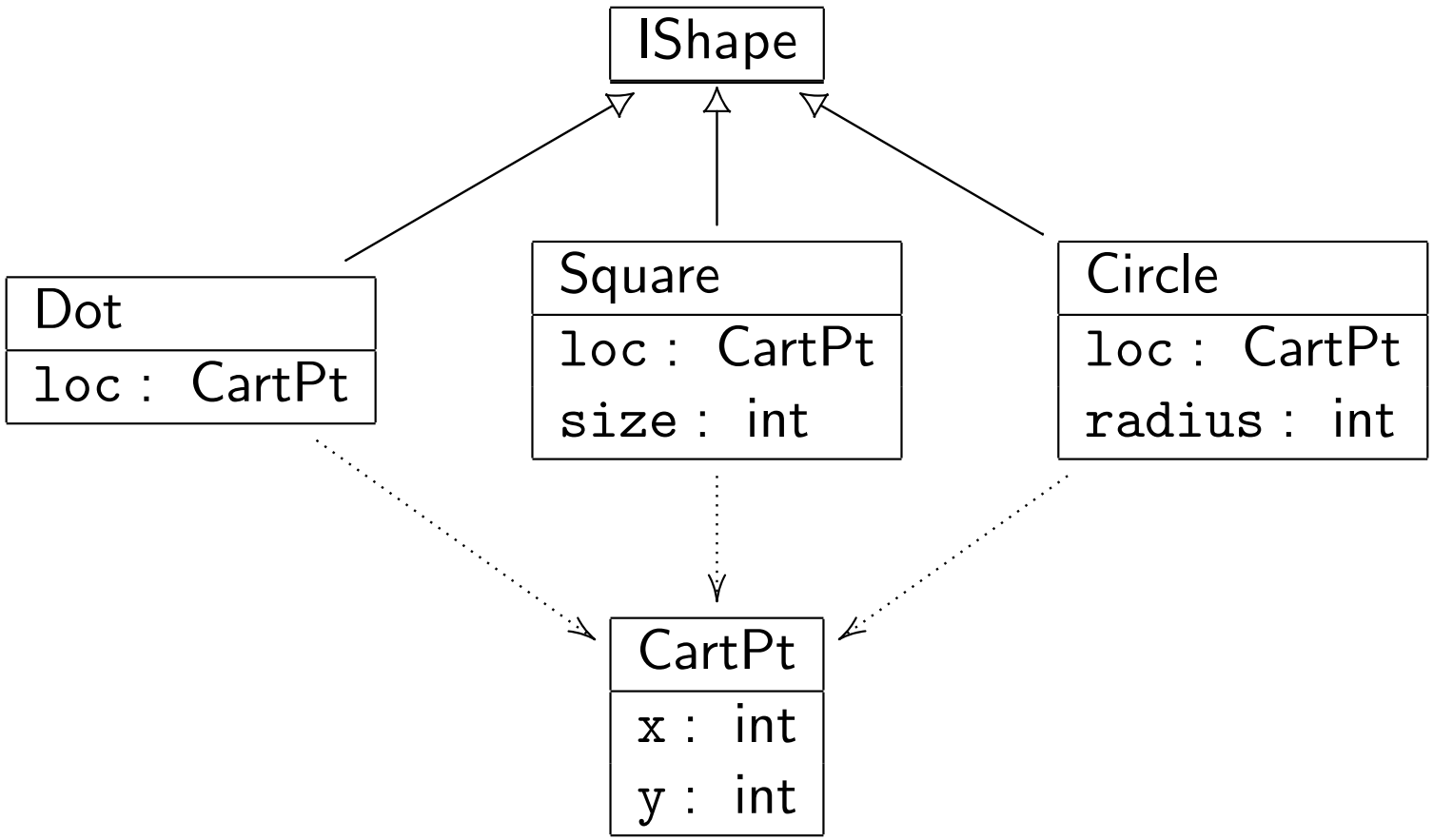
Erstellen einer Vereinigung von Klassen

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

Methoden auf Vereinigungen von Klassen

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



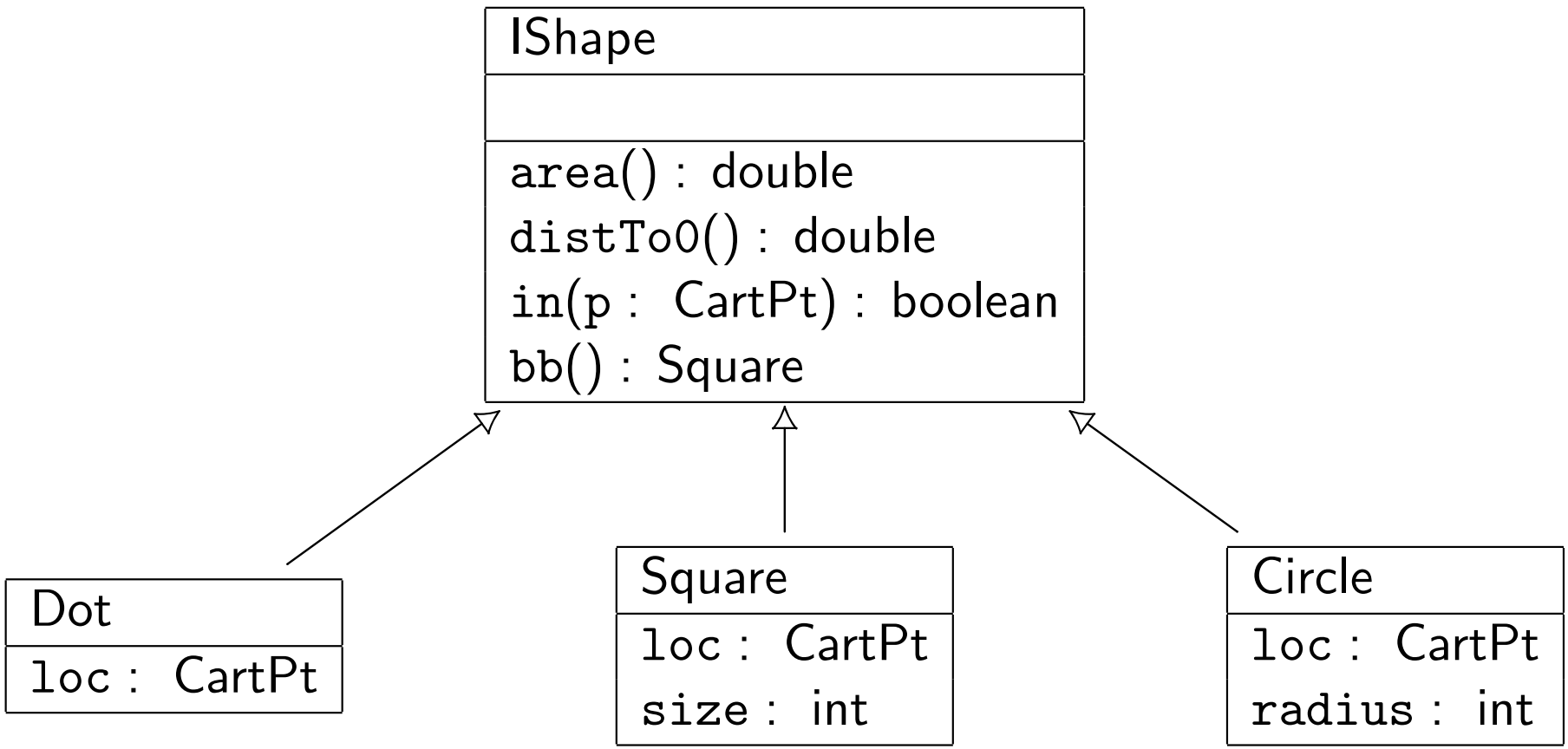
Methoden für IShape

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

```
1  class ShapeExamples {
2      IShape dot = new Dot (new CartPt (4,3));
3      IShape squ = new Square (new CartPt (4,3), 3);
4      IShape cir = new Circle (new CartPt (12,5), 2);
5      // tests
8      boolean testDot1 = check dot.area() expect 0.0 within 0.1;
9      boolean testSqu1 = check squ.area() expect 9.0 within 0.1;
10     boolean testCir1 = check cir.area() expect 12.56 within 0.01;
23     // constructor
24     ShapeExamples () {}
25 }
```

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 für `distTo0()`

```
1  class ShapeExamples {
2      IShape dot = new Dot (new CartPt (4,3));
3      IShape squ = new Square (new CartPt (4,3), 3);
4      IShape cir = new Circle (new CartPt (12,5), 2);
5      // tests
13     boolean testDot2 = check dot.distTo0() expect 5.0 within 0.01;
14     boolean testSqu2 = check squ.distTo0() expect 5.0 within 0.01;
15     boolean testCir2 = check cir.distTo0() expect 11.0 within 0.01;
23     // constructor
24     ShapeExamples () {}
25 }
```

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 nicht den Ursprung enthält.)

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 nicht den Ursprung enthält.)

$\Rightarrow$ Die Hilfsmethode auf CartPt muss selbst den Abstand zum Ursprung berechnen:

$\Rightarrow$ 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 this.loc.distTo0() − this.radius;
}
```

- 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 für bb()

```
1  class ShapeExamples {
2      IShape dot = new Dot (new CartPt (4,3));
3      IShape squ = new Square (new CartPt (4,3), 3);
4      IShape cir = new Circle (new CartPt (12,5), 2);
5      // tests
18     boolean testDot3 = check dot.bb() expect new Square (new CartPt (4,3), 1);
19     boolean testSqu3 = check squ.bb() expect squ;
20     boolean testCir3 = check cir.bb() expect new Square (new CartPt (10,3), 4);
23     // constructor
24     ShapeExamples () {}
25 }
```

Einzige Schwierigkeit

Implementierung für Circle, wo ein Quadrat konstruiert werden muss, dass 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

Die Methode `bb()` ist für einen Punkt 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 in der IDE über die Methode `Util.error(String message)` ausgelöst werden können.

```
Square bb() {
  return Util.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.