Programmierzertifikat Objekt-Orientierte Programmierung mit Java

Vorlesung 02: Funktionale Methoden

Peter Thiemann

Universität Freiburg, Germany

SS 2008

Inhalt

Funktionale Methoden

- Ausdrücke mit primitiven Datentypen
- Methoden entwerfen
- Methoden mit Fallunterscheidung
- Methoden und zusammengesetzte Objekte
- Methoden auf Vereinigungen von Klassen
- Methoden auf rekursiven Klassen

Ausdrücke mit primitiven Datentypen

int, double, boolean

- Ausdrücke dienen zur Berechnung von neuen Werten
- Für primitive Datentypen sind viele *Infixoperatoren* vordefiniert, die auf die gewohnte Art verwendet werden.
  - `60 * .789`
  - `this.x + 2`
  - `Math.PI * radius`

Bemerkungen

- `this.x` liefert den Wert der Instanzvariable `x`
- `Math.PI` liefert den vordefinierten Wert von π (als Gleitkommazahl)

Ausdrücke mit mehreren Operatoren

Für die Operatoren gelten die üblichen Präzedenzregeln (Punkt- vor Strichrechnung usw.)

- `5 * 7 + 3` entspricht `(5 * 7) + 3`
- `position > 0 && position <= maxpos` entspricht `(position > 0) && (position <= maxpos)`

Empfehlung

Verwende generell Klammern

Arithmetische und logische Operatoren (Auszug)

Symbol	Parameter	Ergebnis	Beispiel	
!	boolean	boolean	`!true`	logische Negation
&&	boolean, boolean	boolean	`a && b`	logisches Und
||	boolean, boolean	boolean	`a || b`	logisches Oder
+	numerisch, numerisch	numerisch	`a + 7`	Addition
-	numerisch, numerisch	numerisch	`a - 7`	Subtraktion
*	numerisch, numerisch	numerisch	`a * 7`	Multiplikation
/	numerisch, numerisch	numerisch	`a / 7`	Division
%	numerisch, numerisch	numerisch	`a % 7`	Modulo
<	numerisch, numerisch	boolean	`y < 7`	kleiner als
<=	numerisch, numerisch	boolean	`y <= 7`	kleiner gleich
>	numerisch, numerisch	boolean	`y > 7`	größer als
>=	numerisch, numerisch	boolean	`y >= 7`	größer gleich
==	numerisch, numerisch	boolean	`y == 7`	gleich
!=	numerisch, numerisch	boolean	`y != 7`	ungleich

Der primitive Typ String

- String ist vordefiniert, ist aber ein Klassentyp
  d.h. jeder String ist ein Objekt
- Ein Infixoperator ist definiert:

Symbol	Parameter	Ergebnis	Beispiel	
+	String, String	String	`s1 + s2`	Stringverkettung

```
"laber" + "fasel" // ==> "laberfasel"
```

- Wenn einer der Parameter numerisch oder boolean ist, so wird er automatisch in einen String *konvertiert*.

```
"x=" + 5 // ==> "x=5"
"this is " + false // ==> "this is false"
```

Methodenaufrufe

Methoden von String

- Weitere Stringoperation sind als *Methoden* der Klasse String definiert und durch *Methodenaufrufe* verfügbar.
- Beispiele
  - `"arachnophobia".length()` Stringlänge
  - `"wakarimasu".concat ("ka")` Stringverkettung
- Allgemeine Form
  eObject`.`*methode*`(`*eArg*`, ...)`
  - *eObject* Ausdruck, dessen Wert ein Objekt ist
  - *methode* Name einer Methode dieses Objektes
  - *eArg* Argumentausdruck für die Methode
- Schachtelung möglich (Auswertung von links nach rechts)
  `"mai".concat("karenda").length()`

Einige String Methoden

Methode	Parameter	Ergebnis	Beispiel
`length`	()	int	`"xy".length()`
`concat`	(String)	String	`"xy".concat ("zw")`
`toLowerCase`	()	String	`"XyZ".toLowerCase()`
`toUpperCase`	()	String	`"XyZ".toUpperCase()`
`equals`	(String)	boolean	`"XyZ".equals ("xYz")`
`endsWith`	(String)	boolean	`"XyZ".endsWith("yZ")`
`startsWith`	(String)	boolean	`"XyZ".startsWith("Xy")`

- Insgesamt 54 Methoden (vgl. java.lang.String)

Methoden entwerfen

Objekte erhalten ihre Funktionalität durch *Methoden*

Beispiel

Zu einer Teelieferung (bestehend aus Teesorte, Kilopreis und Gewicht) soll der Gesamtpreis bestimmt werden.

- Implementierung durch Methode `cost()`
- Keine Parameter, da alle Information im Tea-Objekt vorhanden ist.
- Ergebnis ist ein Preis, repräsentiert durch den Typ int
- Verwendungsbeispiel:
  ```
  Tea tAssam = new Tea("Assam", 2790, 150);
  tAssam.cost()
  ```
  soll 418500 liefern

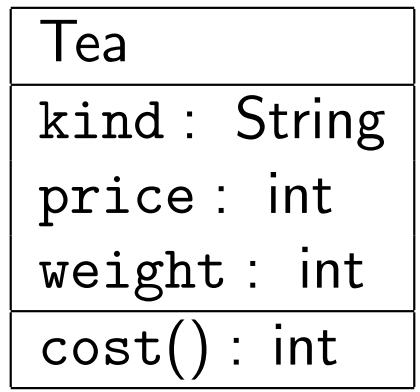
Methodendefinition

```
// Repräsentation einer Rechnung für eine Teelieferung
class Tea {
    String kind; // Teesorte
    int price; // in Eurocent pro kg
    int weight; // in kg

    // Konstruktor (wie vorher)
    Tea (String kind, int price, int weight) { ... }

    // berechne den Gesamtpreis dieser Lieferung
    int cost() { ... }

}
```

- Methodendefinitionen nach Konstruktor
- Methode `cost()`
  - Ergebnistyp int
  - keine Parameter
  - Rumpf muss jetzt ausgefüllt werden

Klassendiagramm mit Methoden

Gleiche Information im Klassenkasten



- Dritte Abteilung enthält die Kopfzeilen der Methoden *Signaturen von Methoden*

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

Entwicklung der Methode `cost`

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

- Zugriff auf die Felder des Objekts erfolgt mittels `this.`*`feldname`*

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(`kind` spielt hier keine Rolle)

Entwicklung der Methode cost

- Jede Methode kann auf ihr zugehöriges Objekt über die Variable `this` zugreifen

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this ... }
```

- Zugriff auf die Felder des Objekts erfolgt mittels `this.`*`feldname`*

```
// berechne den Gesamtpreis dieser Lieferung
int cost() { ... this.kind ... this.price ... this.weight ... }
```

(`kind` spielt hier keine Rolle)

- Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
18  // berechne den Gesamtpreis dieser Lieferung
19  int cost() {
20      return this.price * this.weight;
21  }
```

Methodentest

```
1  // Tests für die Methoden von Tea
2  class TeaExamples {
3    Tea tea1 = new Tea("Assam", 1590, 150);
4    Tea tea2 = new Tea("Darjeeling", 2790, 220);
5    Tea tea3 = new Tea("Ceylon", 1590, 130);
6
7    boolean testTea1 = check this.tea1.cost() expect 238500;
8    boolean testTea2 = check this.tea2.cost() expect 613800;
9    boolean testTea3 = check this.tea3.cost() expect 206700;
10
11   TeaExamples() { }
12 }
```

- Separate Testklasse
- Felder, deren Namen mit `test...` beginnen, werden von der IDE als Tests registriert
- Der `check`-Ausdruck "**check** *expr1* **expect** *expr2*" liefert `true`, falls beide Ausdrücke den gleichen Wert haben (Eine Java-Erweiterung)

Methoden mit Argumenten

Primitive Datentypen

Der Teelieferant sucht nach Billigangeboten, bei denen der Kilopreis kleiner als eine vorgegebene Schranke ist.

- Argumente von Methoden werden wie Felder deklariert

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Gewünschtes Verhalten:

```
check new Tea ("Earl Grey", 3945, 75).cheaperThan (2000) expect false
check new Tea ("Ceylon", 1590, 400).cheaperThan (2000) expect true
```

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(`kind` und `weight` spielen hier keine Rolle)

Methoden mit Argumenten

Primitive Datentypen/2

- Methodensignatur

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this ... }
```

- Im Rumpf der Methode dürfen die Felder des Objekts und die Parameter verwendet werden.

```
// liegt der Kilopreis dieser Lieferung unter limit?
boolean cheaperThan(int limit) { ... this.price ... limit ... }
```

(`kind` und `weight` spielen hier keine Rolle)

- Der Rückgabewert der Methode wird durch die **return**-Anweisung spezifiziert.

```
24  boolean cheaperThan(int limit) {
25      return this.price < limit;
26  }
```

Methoden mit Argumenten

Klassentypen

Der Teelieferant möchte Lieferungen nach ihrem Gewicht vergleichen.

- Argumente von Methoden werden wie Felder deklariert

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

- Gewünschtes Verhalten:

```
Tea t1 = new Tea ("Earl Grey", 3945, 75);
Tea t2 = new Tea ("Ceylon", 1590, 400);

check t1.lighterThan (new Tea ("Earl Grey", 3945, 25)) expect false
check t2.cheaperThan (new Tea ("Assam", 1590, 500)) expect true
```

Methoden mit Argumenten

Klassentypen/2

- Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

Methoden mit Argumenten

Klassentypen/2

- Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

- Alle Felder beider Objekte sind verwendbar:

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) {
  ... this.kind ... that.kind ...
  ... this.price ... that.price ...
  ... this.weight ... that.weight ...
}
```

Methoden mit Argumenten

Klassentypen/2

- Methodensignatur

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) { ... this ... that ... }
```

- Alle Felder beider Objekte sind verwendbar:

```
// wiegt diese Lieferung mehr als eine andere?
boolean lighterThan(Tea that) {
  ... this.kind ... that.kind ...
  ... this.price ... that.price ...
  ... this.weight ... that.weight ...
}
```

- Der Methodenrumpf verwendet das Feld `weight` von beiden Objekten

```
29  boolean lighterThan(Tea that) {
30      return this.weight < that.weight;
31  }
```

Rezept für den Methodenentwurf

Ausgehend von einer Klasse

1. erkläre kurz den Zweck der Methode (Kommentar)
2. definiere die Methodensignatur
3. gib Beispiele für die Verwendung der Methode
4. fülle den Rumpf der Methode gemäß dem Muster
   - **this** und die Felder **this**.*feldname* dürfen vorkommen
   - alle Parameter dürfen vorkommen
5. schreibe den Rumpf der Methode
6. definiere die Beispiele als Tests

Methoden mit Fallunterscheidung

Eine Bank verzinst eine Spareinlage jährlich mit einem gewissen Prozentsatz. Der Prozentsatz hängt von der Höhe der Einlage ab. Unter 5000 Euro gibt die Bank 4,9% Zinsen, bis unter 10000 Euro 5,0% und für höhere Einlagen 5,1%. Berechne den Zins für eine Einlage.

- Klassendiagramm dazu

Deposit
`owner` : String `amount [in Cent]` : int
`interest()` : double

Analyse der Zinsberechnung

- Beispiele

```
check new Deposit ("Dieter", 120000).interest() expect 5880.0
check new Deposit ("Verona", 500000).interest() expect 25000.0
check new Deposit ("Franjo", 1100000).interest() expect 56100.0
```

- In der Methode `interest` gibt es drei Fälle, die von dem Betrag **this**.`amount` abhängen.
- Die drei Fälle werden mit einer *bedingten Anweisung* (If-Anweisung) unterschieden.

Bedingte Anweisung

- Allgemeine Form

```
if (bedindung) {
  anweisung1 // ausgeführt, falls bedingung wahr
} else {
  anweisung2 // ausgeführt, falls bedingung falsch
}
```

- Zur Zeit kennen wir nur die **return**-Anweisung

```
if (bedindung) {
  return ausdruck1; // ausgeführt, falls bedingung wahr
} else {
  return ausdruck2; // ausgeführt, falls bedingung falsch
}
```

Bedingte Anweisung geschachtelt

- Die bedingte Anweisung ist selbst eine Anweisung, also ist auch Schachtelung möglich.

```
if (bedindung1) {
  // ausgeführt, falls bedingung1 wahr
  return ausdruck1;
} else {
  if (bedindung2) {
    // ausgeführt, falls bedingung1 falsch und bedingung2 wahr
    return ausdruck2;
  } else {
    // ausgeführt, falls bedingung1 und bedingung2 beide falsch
    return ausdruck3;
  }
}
```

$\Rightarrow$ Passt genau zu den Anforderungen an `interest()`!

Bedingte Anweisung für Zinsberechnung

- Einsetzen der Bedingungen und aufführen der verfügbaren Variablen liefert

```
// berechne den Zinsbetrag für diese Objekt
double interest () {
  if (this.amount < 500000) {
    // ausgeführt, falls Betrag < 5000 Euro
    return ... this.owner ... this.amount ... ;
  } else {
    if (this.amount < 1000000) {
      // ausgeführt, falls Betrag >= 5000 Euro und < 10000 Euro
      return ... this.owner ... this.amount ...;
    } else {
      // ausgeführt, falls Betrag >= 10000 Euro
      return ... this.owner ... this.amount ...;
    }
  }
}
```

Methode für Zinsberechnung

- Einsetzen der Zinssätze und der Zinsformel liefert

```
// berechne den Zinsbetrag für diese Objekt
double interest () {
  if (this.amount < 500000) {
    // ausgeführt, falls Betrag < 5000 Euro
    return this.amount * 4.9 / 100 ;
  } else {
    if (this.amount < 1000000) {
      // ausgeführt, falls Betrag >= 5000 Euro und < 10000 Euro
      return this.amount * 5.0 / 100;
    } else {
      // ausgeführt, falls Betrag >= 10000 Euro
      return this.amount * 5.1 / 100;
    }
  }
}
```

Verbesserte Zinsberechnung

- Nachteil der `interest()`-Methode:
  Verquickung der Berechnung des Zinssatzes mit der Fallunterscheidung
- Dadurch taucht die Zinsformel 3-mal im Quelltext auf
- Besser: Kapsele die Zinsformel und die Fallunterscheidung jeweils in einer eigenen Methode

Deposit
`owner : String` `amount [in Cent] : int`
`rate() : double` `payInterest() : double`

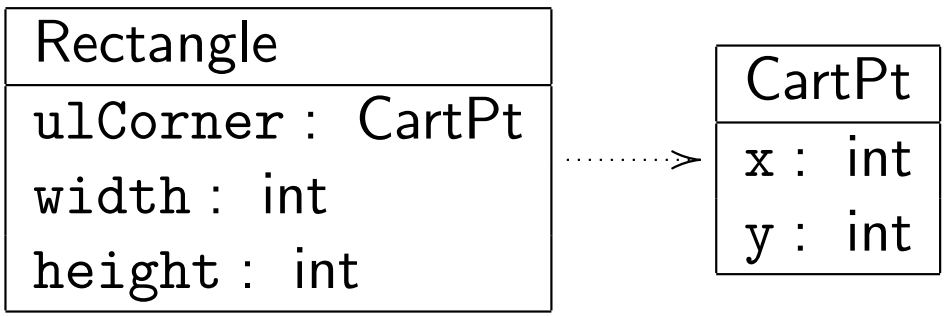
Interner Methodenaufruf

```
27      // bestimme den Zinssatz aus der Höhe der Einlage
28      double rate () {
29          if (this.amount < 500000) {
30              return 4.9;
31          } else {
32              if (this.amount < 1000000) {
33                  return 5.0;
34              } else {
35                  return 5.1;
36              }
37          }
38      }
41      // berechne den Zinsbetrag
42      double payInterest() {
43          return this.amount * this.rate() / 100;
44      }
```

- Die Methoden einer Klasse können sich gegenseitig aufrufen.

Methoden und zusammengesetzte Objekte

- 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);
}
```

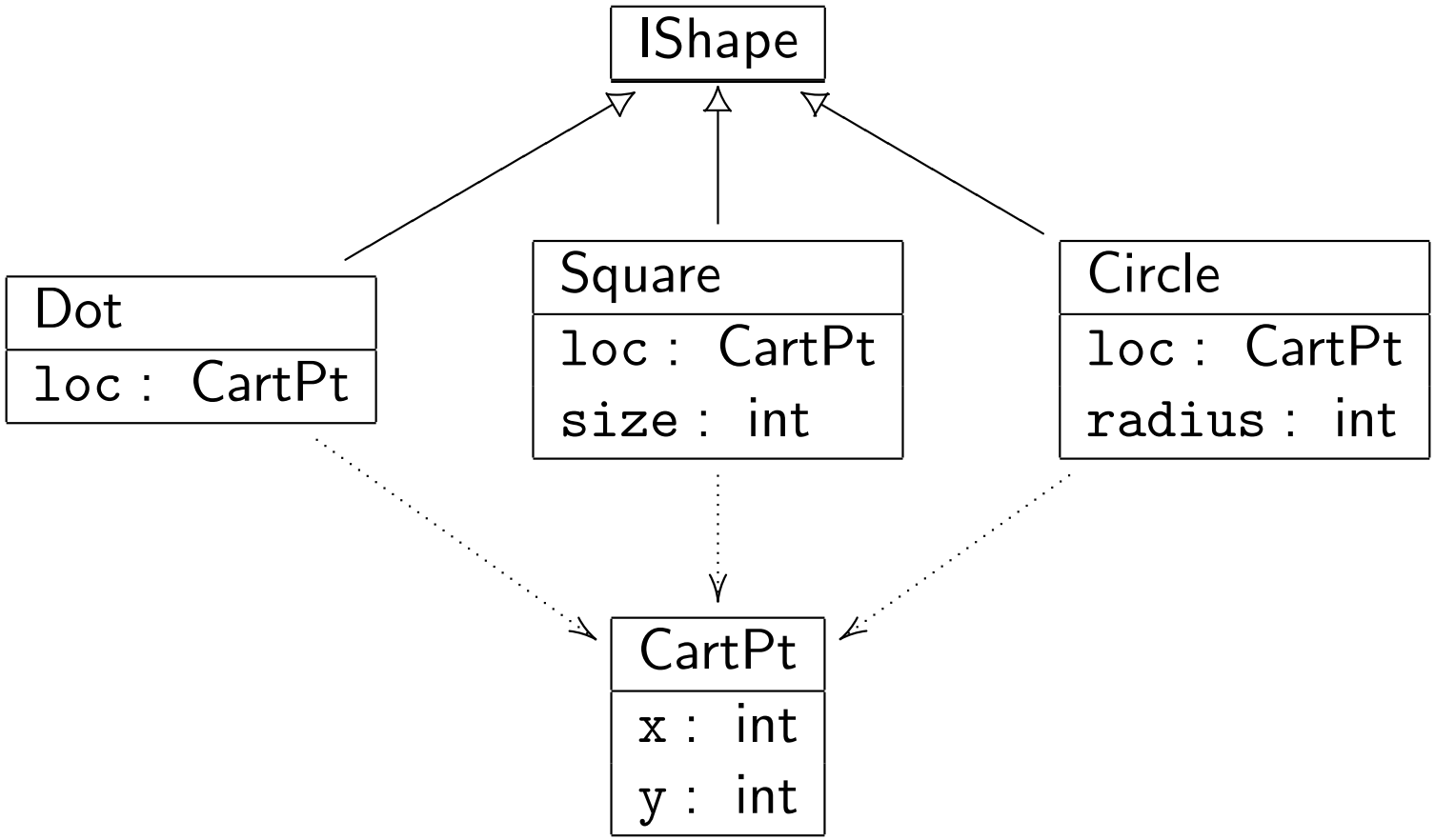
Muster zur Delegation

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.

Methoden auf Vereinigungen von Klassen

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



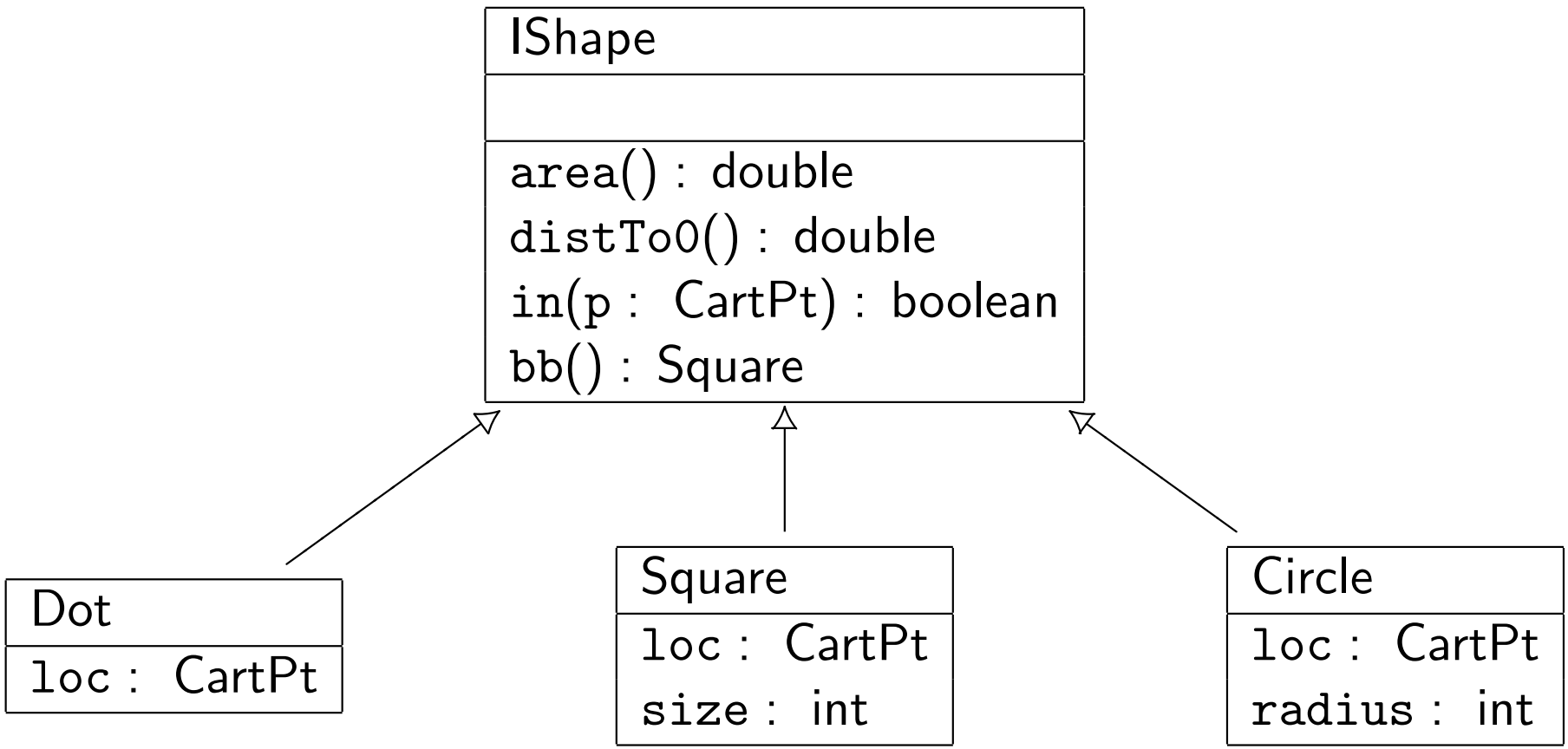
Methoden für IShape

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

```
2  class ShapeExamples {
3      IShape dot = new Dot (new CartPt (4,3));
4      IShape squ = new Square (new CartPt (4,3), 3);
5      IShape cir = new Circle (new CartPt (12,5), 2);
6      // tests
9      boolean testDot1 = check dot.area() expect 0.0 within 0.1;
10     boolean testSqu1 = check squ.area() expect 9.0 within 0.1;
11     boolean testCir1 = check cir.area() expect 12.56 within 0.01;
24     // constructor
25     ShapeExamples () {}
26 }
```

Bemerkung

Beim Rechnen mit double können Rundungsfehler auftreten, so dass ein Test auf Gleichheit nicht angemessen ist. **check_expect_within_** testet daher nicht auf Gleichheit mit dem erwarteten Wert, sondern ob die beiden Werte innerhalb einer Fehlerschranke übereinstimmen.

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()`

```
2  class ShapeExamples {
3      IShape dot = new Dot (new CartPt (4,3));
4      IShape squ = new Square (new CartPt (4,3), 3);
5      IShape cir = new Circle (new CartPt (12,5), 2);
6      // tests
14     boolean testDot2 = check dot.distTo0() expect 5.0 within 0.01;
15     boolean testSqu2 = check squ.distTo0() expect 5.0 within 0.01;
16     boolean testCir2 = check cir.distTo0() expect 11.0 within 0.01;
24     // constructor
25     ShapeExamples () {}
26 }
```

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()

```
2  class ShapeExamples {
3      IShape dot = new Dot (new CartPt (4,3));
4      IShape squ = new Square (new CartPt (4,3), 3);
5      IShape cir = new Circle (new CartPt (12,5), 2);
6      // tests
19     boolean testDot3 = check dot.bb() expect new Square (new CartPt (4,3), 1);
20     boolean testSqu3 = check squ.bb() expect squ;
21     boolean testCir3 = check cir.bb() expect new Square (new CartPt (10,3), 4);
24     // constructor
25     ShapeExamples () {}
26 }
```

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.

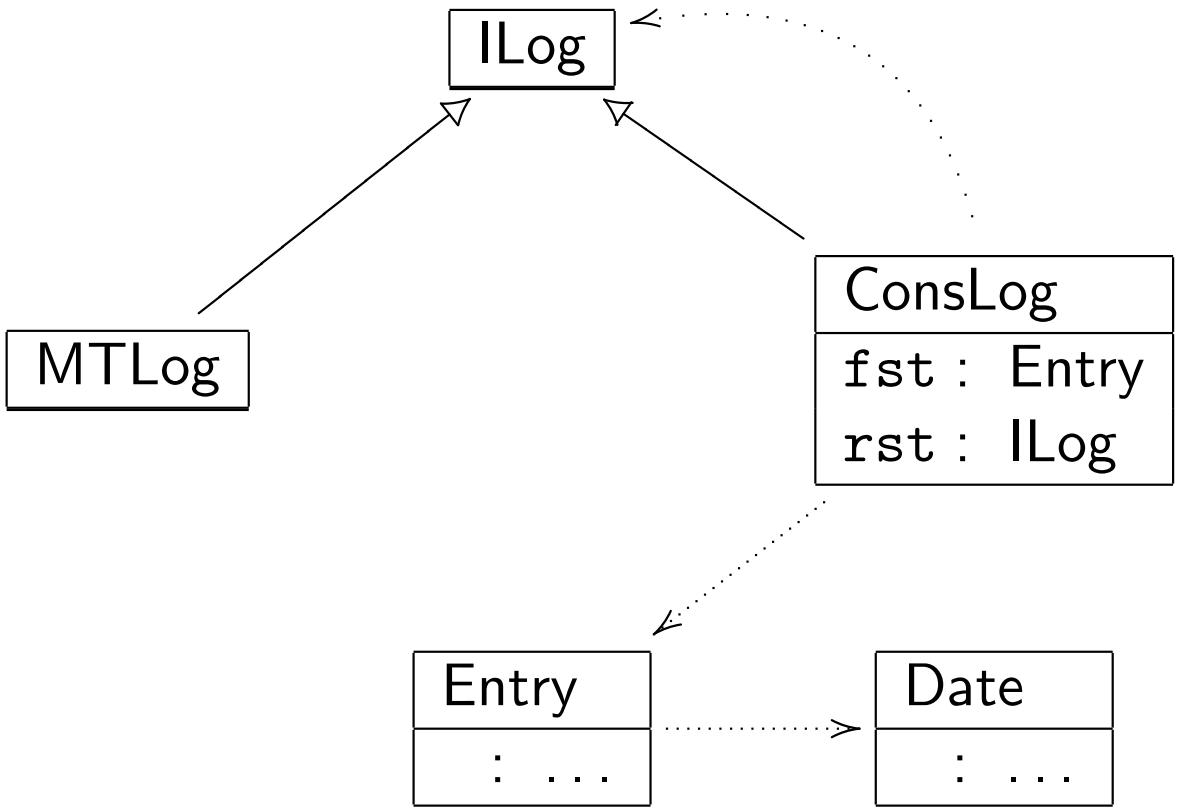
```
Square bb() {
  return Util.error ("bounding box for a dot");
}
```

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.

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() ... // wichtigster (rekursiver) Aufruf
}
```

- ggf. Hilfsmethode in Entry

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

- ggf. Hilfsmethode in Date

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

Beispiel: Gelaufene Strecke

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: Gelaufene Strecke

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: Gelaufene Strecke

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: Gelaufene Strecke

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();
}
```

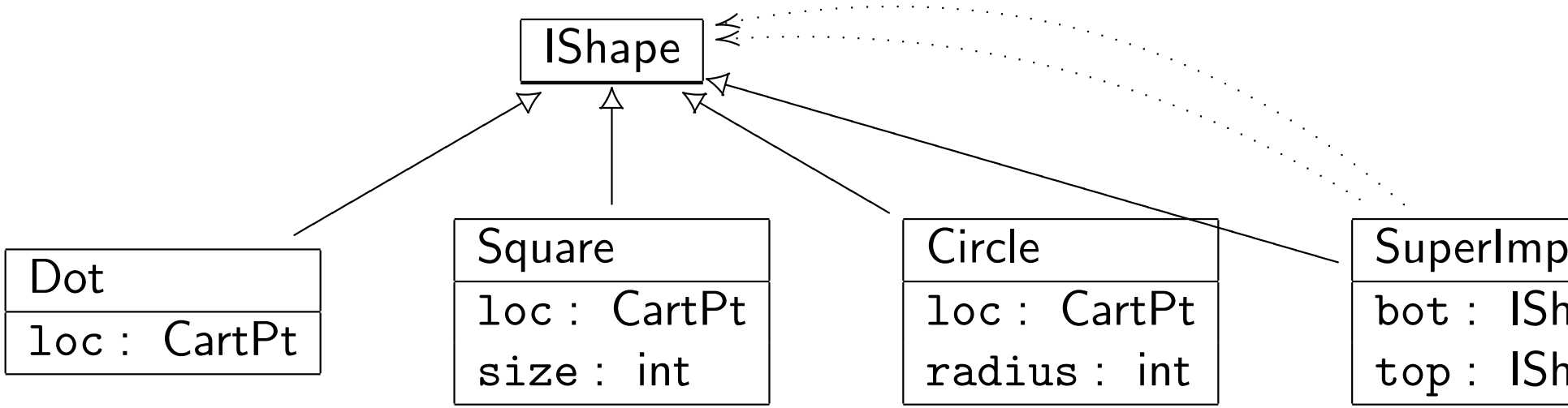
Beispiel: Gelaufene Strecke

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.

Erweiterung von IShape um Überlappung

Erinnerung an 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.

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.