Vorlesung 04: Abstraktion mit Klassen

Peter Thiemann

Universität Freiburg, Germany

SS 2011

Inhalt

Abstraktion mit Klassen

- Abstraktion in der Programmierung bedeutet
  - Auffinden von Mustern
    Wiederkehrende Programmstücke mit ähnlicher Bedeutung
  - Generalisierung
    Ersetzen der spezifischen Teile durch Variable (o.ä.)
  - Extraktion
    Generalisiertes Programmstück benennen und an den ursprünglichen Stellen "aufrufen"
- Allgemein: Vermeiden von duplizierter Information / Arbeit / Fehlern

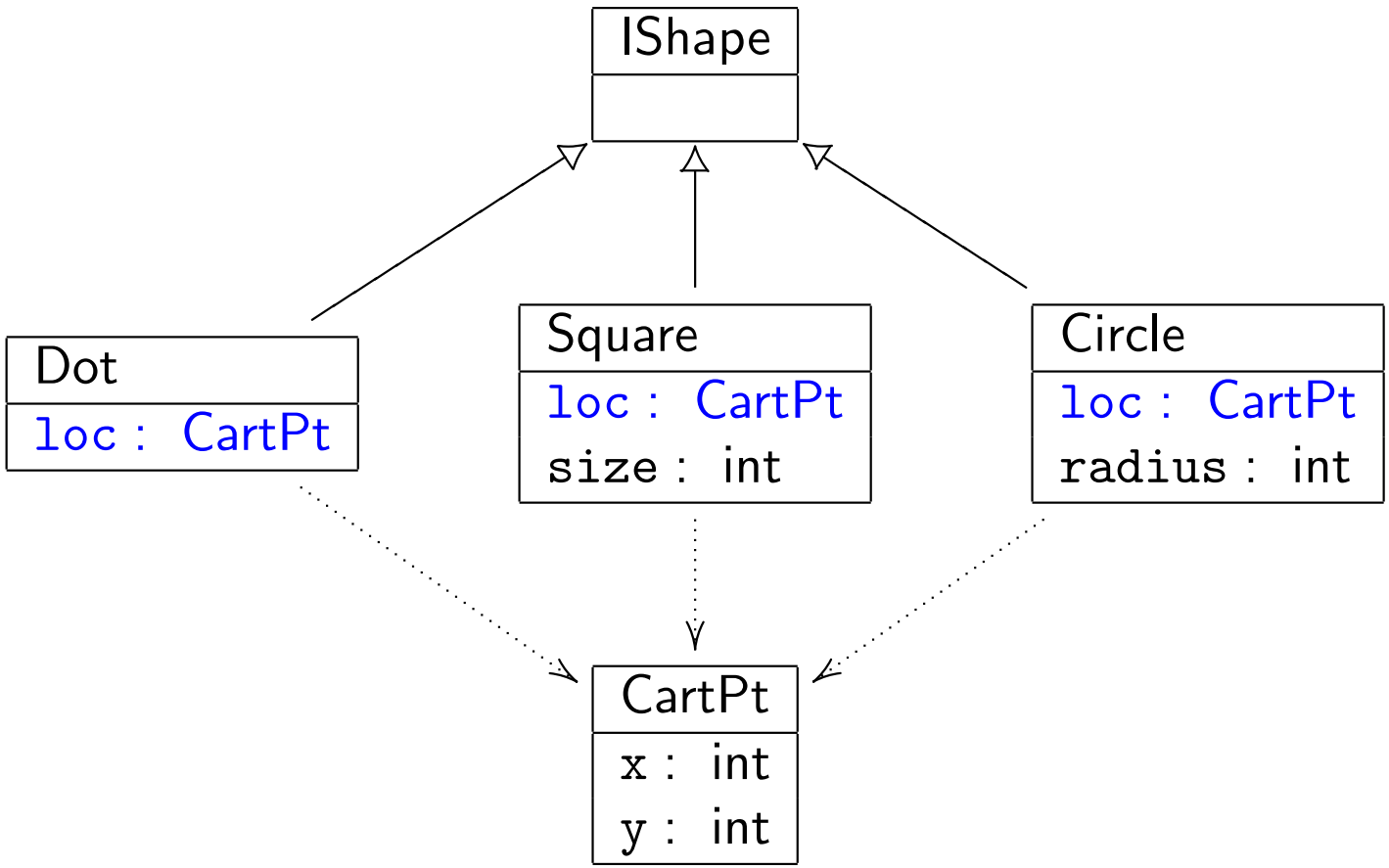
$\Rightarrow$ Suche nach Ähnlichkeiten

$\Rightarrow$ Löse jedes Problem nur einmal

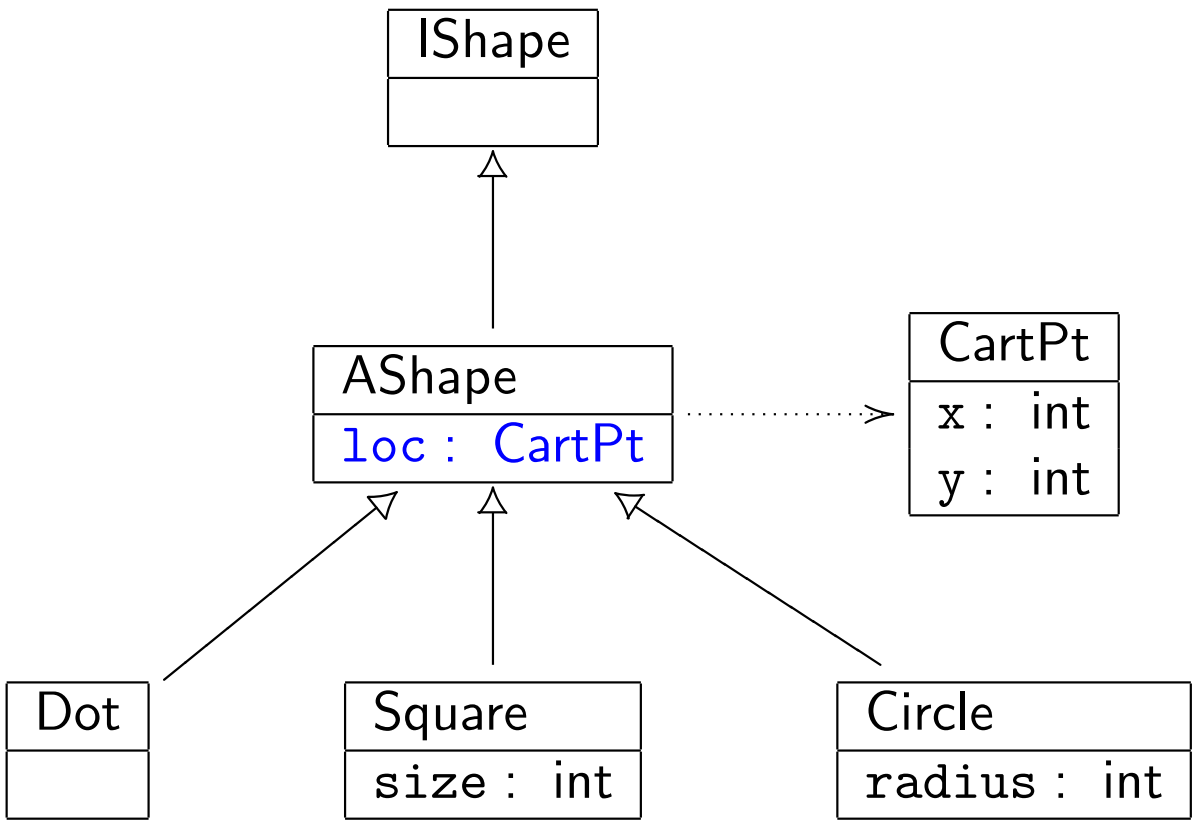
$\Rightarrow$ "Refactoring". Implementiert in Eclipse.

Ähnlichkeiten zwischen Klassen

- Ziel:
  - Identifiziere ähnliche Felder und Methodendefinitionen in Vereinigungen von Klassen
  - Sammle diese Felder in *Superklassen*
- Beispiel: Die IShape-Hierarchie



Extraktion des Bezugspunktes



Einbinden von AShape in die Klassenhierarchie

Implementierung

- IShape bleibt unverändert

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

- Die neue Klasse AShape implementiert IShape

```
// Gemeinsamkeiten aller Figuren
class AShape implements IShape {
    CartPt loc;
    AShape (CartPt loc) {
        this.loc = loc;
    }
}
```

Einbinden von AShape in die Klassenhierarchie

Vererbung

- Alle drei Klassen müssen durch Verwendung des Schlüsselworts **extends** angeben, dass sie von AShape *erben*.
- Sie werden dadurch *Subklassen* von AShape (und AShape ist ihre *Superklasse*)

```
// ein Punkt
class Dot
 extends AShape {
```

```
// ein Quadrat
class Square
 extends AShape {
```

```
// ein Kreis
class Circle
 extends AShape {
```

Vererbung und Konstruktoren

- Der Konstruktor der AShape-Klasse erhält und initialisiert nur das `loc`-Feld.
- Ein Square-Objekt enthält ein `loc`-Feld und ein `size`-Feld.
- Der Konstruktor von Square erhält beide.
- Wie werden sie gesetzt?

Vererbung und Konstruktoren

- Der Konstruktor der AShape-Klasse erhält und initialisiert nur das `loc`-Feld.
- Ein Square-Objekt enthält ein `loc`-Feld und ein `size`-Feld.
- Der Konstruktor von Square erhält beide.
- Wie werden sie gesetzt?
- Delegiere die Verantwortung für `loc` an den AShape-Konstruktor

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

- Der Aufruf `super(loc)` ruft den Konstruktor der Superklasse auf. Er **muss** zu Beginn des Konstruktors der Subklasse verwendet werden.

Vollständige Definition der Subklassen

```
// ein Punkt
class Dot
 extends AShape {
 Dot (CartPt loc) {
  super(loc);
 }
}
```

```
// ein Quadrat
class Square
 extends AShape {
 int size;
 Square (CartPt loc, int size) {
  super(loc);
  this.size = size;
 }
}
```

Vollständige Definition der Subklassen

Fortsetzung

```
// ein Kreis
class Circle
 extends AShape {
 int radius;
 Circle(CartPt loc, int radius) {
  super(loc);
  this.radius = radius;
 }
}
```

Zusammenfassung

1. IShape ist das Interface, das die Funktionalität aller geometrischen Figuren spezifiziert.
2. AShape ist eine Klasse, die die gemeinsamen Attribute aller geometrischen Figuren repräsentiert.
3. Dot, Square und Circle sind *Verfeinerungen* von AShape (Subklassen, *abgeleitete Klassen*). Sie *erben* alle Felder von AShape und **müssen alle Auflagen von IShape erfüllen**.
4. Die Konstruktoren dieser Klassen akzeptieren die initialen Werte aller Felder und übertragen die Initialisierung der gemeinsamen Felder über den `super(...)`-Aufruf an die Superklasse.

Abstrakte Klassen und abstrakte Methoden

- Wenn IShape Methoden spezifiziert und AShape **implements** IShape, dann müssen all diese Methoden auch in AShape definiert werden!
- Beispielmethoden: `area()`, `distTo0()`, `in()` und `bb()`
- Problem: Wie wird (z.B.) `area()` in AShape definiert? Dort ist noch nicht klar, um welche Figur es geht und jede Figure definiert diese Methode anders!

Abstrakte Klassen und abstrakte Methoden

- Wenn IShape Methoden spezifiziert und AShape **implements** IShape, dann müssen all diese Methoden auch in AShape definiert werden!
- Beispielmethoden: `area()`, `distTo0()`, `in()` und `bb()`
- Problem: Wie wird (z.B.) `area()` in AShape definiert? Dort ist noch nicht klar, um welche Figur es geht und jede Figure definiert diese Methode anders!
- Antwort: Diese Methoden werden in AShape als *abstrakte Methoden* definiert, aber **nicht** implementiert.
- Jede Subklasse von AShape **muss** eine Implementierung für alle abstrakten Methoden bereitstellen.

Abstrakte Klassen und abstrakte Methoden

- Wenn IShape Methoden spezifiziert und AShape **implements** IShape, dann müssen all diese Methoden auch in AShape definiert werden!
- Beispielmethoden: `area()`, `distTo0()`, `in()` und `bb()`
- Problem: Wie wird (z.B.) `area()` in AShape definiert? Dort ist noch nicht klar, um welche Figur es geht und jede Figure definiert diese Methode anders!
- Antwort: Diese Methoden werden in AShape als *abstrakte Methoden* definiert, aber **nicht** implementiert.
- Jede Subklasse von AShape **muss** eine Implementierung für alle abstrakten Methoden bereitstellen.
- Eine Klasse mit abstrakten Methoden kann selbst keine Objekte erzeugen, sie wird dadurch zur *abstrakten Klasse*.

IShape und AShape mit Methoden

```
// 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();
}
```

```
// Gemeinsamkeiten aller Figuren
abstract class AShape implements IShape {
    CartPt loc;
    abstract double area();
    abstract double distTo0();
    abstract boolean in(CartPt p);
    abstract Square bb();
}
```

Vollständige Definition der Subklassen

(ohne Konstruktoren)

```
// ein Punkt
class Dot
 extends AShape {
 double area () {
  return 0;
 }
 double distTo0 () {
  return this.loc.distTo0();
 }
 ...
}
```

```
// ein Quadrat
class Square
 extends AShape {
 int size;
 double area() {
  return this.size *
        this.size;
 }
 double distTo0 () {
  return this.loc.distTo0();
 }
 ...
}
```

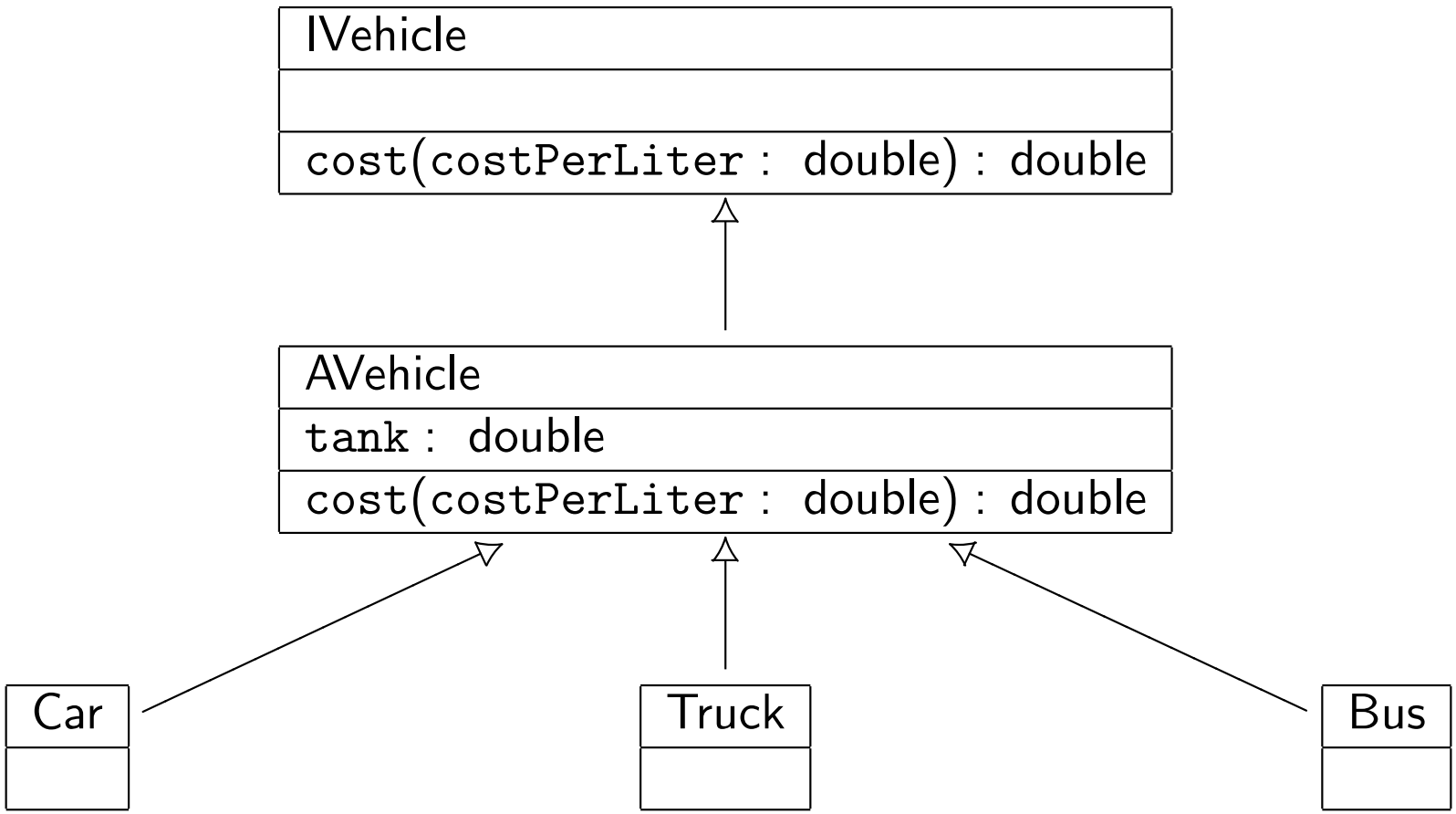
```
// ein Kreis
class Circle
 extends AShape {
 double area () {
  return this.radius *
        this.radius *
        Math.PI;
 }
 double distTo0 () {
  return this.loc.distTo0()
        - radius;
 }
 ...
}
```

Hochheben und Vererben von Methoden

- Methodendefinitionen können genauso vererbt werden wie Felddefinitionen
- Sinnvoll, falls alle Definitionen einer Methode in einer Vereinigung von Klassen identisch sind.

Beispiel: Fahrzeuge

Verschiedene Arten von Fahrzeugen mit einer Methode, die die Kosten eines Tankinhalts berechnet



Beispiel: Methodenimplementierungen nach Konstruktion

- in Car

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

- in Truck

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

- in Bus

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

Beispiel: Methodenimplementierungen nach Konstruktion

- in Car

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

- in Truck

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

- in Bus

```
double cost (double costPerLiter) {
  return this.tank * costPerLiter;
}
```

- Diese Definition kann nach AVehicle hochgehoben und von dort geerbt werden!

Beispiel: Revidierte Definitionen

- in AVehicle

```
abstract class AVehicle implements IVehicle {
  double tank;
  ...
  double cost (double costPerLiter) {
    return this.tank * costPerLiter;
  }
}
```

- Diese Definition von `cost` gilt auch für Car, Truck und Bus und kann dort entfernt werden.

Beispiel: Größenvergleich von Figuren

Versuch 1

```
interface IShape {
  // berechne die Fläche einer Figur
  double area();
  // ist diese Figur größe als eine andere?
  boolean larger(IShape that);
}
```

```
abstract class AShape implements IShape {
  CartPt loc;
  ...
  abstract double area();
  abstract boolean larger (IShape that);
}
```

Beobachtungen über die Implementierung von `larger()`:

- Sie verwendet `area()`
- Sie ist in jeder Subklasse gleich

Beispiel: Größenvergleich von Figuren

Versuch 2

```
interface IShape {
  // berechne die Fläche einer Figur
  double area();
  // ist diese Figur größe als eine andere?
  boolean larger(IShape that);
}
```

```
abstract class AShape implements IShape {
  CartPt loc;
  abstract double area();
  boolean larger (IShape that) {
    return this.area() > that.area();
  }
}
```

Beobachtungen über die Implementierung von `larger()`:

- Verwendet `area()`, obwohl diese erst in Subklassen definiert wird!
- In den Subklassen wird `larger()` **nicht** mehr definiert!

Überschreiben von Methoden

Hochheben von fast überall gleichen Methoden

- `distTo0()` ist nur in Circle anders definiert

```
// ein Punkt
class Dot
 extends AShape {
 double area () {
  return 0;
 }
 double distTo0 () {
  return this.loc.distTo0();
 }
 ...
}
```

```
// ein Quadrat
class Square
 extends AShape {
 int size;
 double area() {
  return this.size *
        this.size;
 }
 double distTo0 () {
  return this.loc.distTo0();
 }
 ...
}
```

```
// ein Kreis
class Circle
 extends AShape {
 double area () {
  return this.radius *
        this.radius *
        Math.PI;
 }
 double distTo0 () {
  return this.loc.distTo0()
        - radius;
 }
 ...
}
```

Überschreiben von `distTo0()`

- `distTo0()` ist nur in Circle anders definiert

$\Rightarrow$ Hochheben der gemeinsamen Definition aus Dot und Square in die abstrakte Superklasse AShape

$\Rightarrow$ Löschen der Definition von `distTo0()` in Dot und Square.

- Die Definition von `distTo0()`in Circle verbleibt. Sie hat **die gleiche Signatur wie in der Superklasse** und *überschreibt* die Definition in AShape.

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

- Diese Definition gilt für alle Circle-Objekte.

Aufrufen der Methode auf **super**

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

- Verbleibende Unschönheit: `this.loc.distTo0()` ist die einzige Abhängigkeit von CartPt in der Circle-Klasse.
- Diese Abhängigkeit kann durch Rückgriff auf die Implementierung in der Superklasse behoben werden:

```
double distTo0() {
  return super.distTo0() − this.radius;
}
```

Ein *Supermethodenaufruf* . . .

Erzeugen von Superklassen und Vereinigungen

- Es kann vorkommen, dass Klassen für unterschiedliche Zwecke sehr ähnliche Strukturen aufweisen.
  - gleiche Typen von Instanzvariablen
  - gleiche Methodensignaturen

  aber ggf. mit unterschiedlichen Namen.

$\Rightarrow$ Umbenennen auf gemeinsame Namenskonvention

$\Rightarrow$ Abstrahieren in gemeinsame Superklasse + Interface um Codeduplikation zu vermeiden.

Beispiel: Wetterdaten

In einem Programm zur Manipulation von Wetter treten Messungen von Temperatur und Luftdruck (ggf. noch Niederschlag) auf. Für jede Messung werden Minimal- und Maximalwerte, sowie der aktuelle Wert gespeichert.

- Betrachte zunächst Temperatur und Luftdruck.
- Das Design der beiden Klassen ist sehr ähnlich.

Temperaturmessungen

```
// Temperaturmessungen [in °Celsius]
class Temperature {
    int high;
    int today;
    int low;
    Temperature (int high, int today, int low) { ... }
    // berechne den Unterschied zum Maximalwert
    int dHigh() { return this.high − this.today; }
    // berechne den Unterschied zum Minimalwert
    int dLow() { return this.today − this.low; }
    // liefere eine Stringrepräsentation
    String asString() {
        return String.valueOf(low)
            .concat("−")
            .concat (String.valueOf(high))
            .concat("°C");
    }
}
```

Druckmessungen

```
// Druckmessungen [in hPa]
class Pressure {
    int high;
    int today;
    int low;
    Pressure (int high, int today, int low) { ... }
    // berechne den Unterschied zum Maximalwert
    int dHigh() { return this.high − this.today; }
    // berechne den Unterschied zum Minimalwert
    int dLow() { return this.today − this.low; }
    // liefere eine Stringrepräsentation
    String asString() {
        return String.valueOf(low)
            .concat("−")
            .concat (String.valueOf(high))
            .concat("hPa"); // einzige Änderung (bis auf Klassennamen)
    }
}
```

Abstraktion: Messung

Erster Versuch

- Offenbar bietet sich hier eine Abstraktion an ...
- Definiere eine neue Superklasse.
- Die ursprünglichen Messklassen werden Subklassen, die die Methode `asString()` auf die naheliegende Weise überschreiben.

Abstraktion: Messung

Superklasse

```
// Messungen
class Recording {
    int high;
    int today;
    int low;
    Recording (int high, int today, int low) { ... }
    // berechne den Unterschied zum Maximalwert
    int dHigh() { return this.high − this.today; }
    // berechne den Unterschied zum Minimalwert
    int dLow() { return this.today − this.low; }
    // liefere eine Stringrepräsentation
    String asString() {
        return String.valueOf(low)
            .concat("−")
            .concat (String.valueOf(high)); // ohne Einheit
    }
}
```

Abstraktion: Messung

Subklassen

```
class Pressure extends Recording {
 Pressure (int high, int today, int low) {
  super (high, today, low);
 }

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

```
class Temperature extends Recording {
 Temperature (int high, int today, int low) {
  super (high, today, low);
 }

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

Abstraktion: Messung

Subklassen

```
class Pressure extends Recording {
 Pressure (int high, int today, int low) {
  super (high, today, low);
 }

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

```
class Temperature extends Recording {
 Temperature (int high, int today, int low) {
  super (high, today, low);
 }

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

- Unschön:
  - Die beiden Implementierungen von `asString()` sind gleich bis auf den String, der den Namen der Einheit definiert.
  - Der Programmierer wird *nicht* gezwungen, die Methode `asString()` zu überschreiben!

Abstraktion: Messung

Subklassen

```
class Pressure extends Recording {
 Pressure (int high, int today, int low) {
  super (high, today, low);
 }

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

```
class Temperature extends Recording {
 Temperature (int high, int today, int low) {
  super (high, today, low);
 }

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

- Unschön:
  - Die beiden Implementierungen von `asString()` sind gleich bis auf den String, der den Namen der Einheit definiert.
  - Der Programmierer wird *nicht* gezwungen, die Methode `asString()` zu überschreiben!
- Abstrahiere den String in ein neues Feld der Superklasse und verschiebe auch den Code dorthin!

Abstraktion #2: Messung

Superklasse

```
// Messungen
class Recording {
    int high;
    int today;
    int low;
    String unit; // neues Feld
    Recording (int high, int today, int low, String unit) { ... }
    // berechne den Unterschied zum Maximalwert
    int dHigh() { return this.high − this.today; }
    // berechne den Unterschied zum Minimalwert
    int dLow() { return this.today − this.low; }
    // liefere eine Stringrepräsentation
    String asString() {
        return String.valueOf(low)
            .concat("−")
            .concat (String.valueOf(high))
            .concat (unit); // mit Einheit
    }
}
```

Abstraktion #2: Messung

Subklassen

```
class Pressure extends Recording {
  Pressure (int high, int today, int low) {
    super (high, today, low, "hPa");
  }
}
```

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

Abstraktion #2: Messung

Subklassen

```
class Pressure extends Recording {
  Pressure (int high, int today, int low) {
    super (high, today, low, "hPa");
  }
}
```

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

- Gut: Jede Subklasse muss eine Einheit spezifizieren.

Abstraktion #2: Messung

Subklassen

```
class Pressure extends Recording {
  Pressure (int high, int today, int low) {
    super (high, today, low, "hPa");
  }
}
```

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

- Gut: Jede Subklasse muss eine Einheit spezifizieren.
- Schlecht: Ein Programm kann direkt den Typ Recording benutzen und so den Unterschied zwischen Pressure und Temperature verwischen.

Abstraktion #2: Messung

Subklassen

```
class Pressure extends Recording {
  Pressure (int high, int today, int low) {
    super (high, today, low, "hPa");
  }
}
```

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

- Gut: Jede Subklasse muss eine Einheit spezifizieren.
- Schlecht: Ein Programm kann direkt den Typ Recording benutzen und so den Unterschied zwischen Pressure und Temperature verwischen.
- Lösung: Die Subklasse muss die Einheit durch eine Methode spezifizieren!

Abstraktion #3: Messung mit *Template and Hook*

Superklasse

```
// Messungen
abstract class ARecording {
    int high;
    int today;
    int low;
    Recording (int high, int today, int low) { ... }
    int dHigh() { ... }
    int dLow() { ... }
    // bestimme die Einheit; muss von Subklasse definiert werden.
    abstract String unit();
    // liefere eine Stringrepräsentation
    String asString() {
        return String.valueOf(low)
            .concat("-")
            .concat (String.valueOf(high))
            .concat (unit()); // mit Einheit
    }
}
```

Abstraktion #3: Messung mit *Template and Hook*

Subklassen

```
class Pressure extends Recording {
 Pressure (int high, int today, int low) {
  super (high, today, low);
 }
 String unit() {
  return "hPa";
 }
}
```

```
class Temperature extends Recording {
 Temperature (int high, int today, int low) {
  super (high, today, low);
 }
 String unit() {
  return "°C";
 }
}
```

Abstraktion #3: Messung mit *Template and Hook*

Subklassen

```
class Pressure extends Recording {
 Pressure (int high, int today, int low) {
  super (high, today, low);
 }
 String unit() {
  return "hPa";
 }
}
```

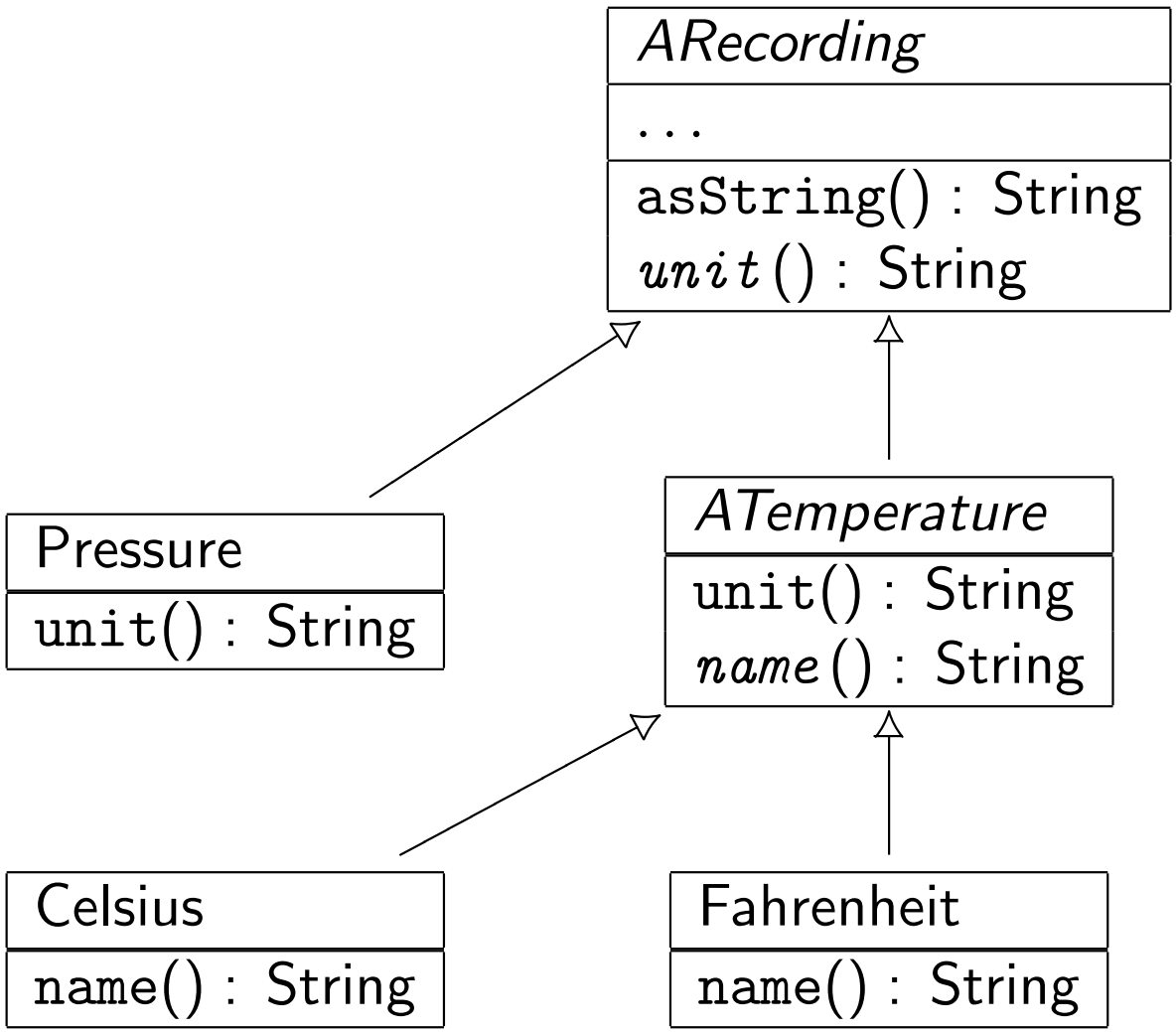
```
class Temperature extends Recording {
 Temperature (int high, int today, int low) {
  super (high, today, low);
 }
 String unit() {
  return "°C";
 }
}
```

- Gut: Jede Subklasse muss eine Einheit spezifizieren.
- Gut: Ein Programm kann keine Objekte vom Typ ARecording erzeugen. Der Unterschied zwischen Pressure und Temperature bleibt aufrechterhalten.
- Gut: Es können Methoden geschrieben werden, die gleichermaßen für Pressure und Temperature wirken.

Das *Template-and-Hook* Muster

- Das Zusammenspiel von `asString()` und `unit()` ist ein Beispiel für das *Template and Hook* Muster.
- Die Methode `asString()` in der Superklasse spielt dabei die Rolle des Template (Schablone) und verwendet `unit()` als Hook (Erweiterungspunkt).
- Das *Template* gibt die grundsätzliche Funktionalität vor und sieht *Hooks* vor, an denen Subklassen erforderliche Erweiterungen anbringen.
- Abstrakte Hookmethoden *zwingen* die Subklasse die Erweiterung durchzuführen.
- Das Template-and-Hook Muster kann mehrfach in einer Klassenhierarchie auftreten.

Mehrfaches Auftreten von Template-and-Hook



Implementierung von ATemperature und Co

```
abstract class ATemperature extends Recording {
 ATemperature (int high, int today, int low) {
  super (high, today, low);
 }
 String unit() {
  return " degrees ".concat(this.name());
 }
 abstract String name();
}
```

```
class Celsius extends ATemperature {
 Celsius (int high, int today, int low) {
  super (high, today, low);
 }
 String name() {
  return "Celsius";
 }
}
```

```
class Fahrenheit extends ATemperature {
 Fahrenheit (int high, int today, int low) {
  super (high, today, low);
 }
 String name() {
  return "Fahrenheit";
 }
}
```

Erweiterung um Niederschlagsmessungen

```
// Niederschlagsmessungen [in mm]
class Precipitation extends ARecording {
    Precipitation (int high, int today) {
        super (high, today, 0);
    }
    // überschreibe asString() um nur einen Maximalwert zu zeigen
    String asString() {
        return "up to ".concat (String.valueOf(high)).concat (this.unit());
    }
    // die unit() Methode muss definiert werden
    String unit() {
        return "mm";
    }
}
```

- Besonderheit: Es gibt einen trivialen Minimalwert, der immer gilt.

Intermezzo: Lokale Variable

Definiere eine Methode für CartPt, *die den Abstand zwischen zwei Punkten* (x_1, x_2) *und* (y_1, y_2) *ausrechnet. Das Ergebnis ist*

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Abstand zwischen zwei Punkten

Direkte Lösung

```
// berechne den Abstand zwischen diesem Punkt und einem anderen
double distance (CartPt that) {
  return Math.sqrt (
  ((this.x − that.x) ∗ (this.x − that.x))
  +
  ((this.y − that.y) ∗ (this.y − that.y)));
}
```

- Nachteil: `(this.x - that.x)` und `(this.y - that.y)` werden mehrmals berechnet

Lösung mit vorhandenen Methoden: Komposition

- Die Methode `distTo0()` kann bereits den Abstand zum Ursprung (d.h., die Länge eines Vektors) ausrechnen.
- Der Abstand zwischen zwei Punkte ist die Länge der Differnz.

```
// berechne den Abstand zwischen diesem Punkt und einem anderen
double distance (CartPt that) {
  return new CartPt (this.x − that.x, this.y − that.y).distTo0();
}
```

Lösung mit vorhandenen Methoden: Komposition

- Die Methode `distTo0()` kann bereits den Abstand zum Ursprung (d.h., die Länge eines Vektors) ausrechnen.
- Der Abstand zwischen zwei Punkte ist die Länge der Differnz.

```
// berechne den Abstand zwischen diesem Punkt und einem anderen
double distance (CartPt that) {
  return new CartPt (this.x − that.x, this.y − that.y).distTo0();
}
```

- Nachteil: Unübersichtlich...

$\Rightarrow$ Wunsch: Benennen der Teilergebnisse

Lokale Variable

- Die Definition einer lokalen Variablen ist eine *Anweisung*.
  typ variable = ausdruck;
- Gültig für die folgenden Anweisungen innerhalb der Methode.
- Vermeidet wiederholte Berechnungen
- Benennt Teilergebnisse
- Sinnvolle Namen dienen der Dokumentation

```
// berechne den Abstand zwischen diesem Punkt und einem anderen
double distance (CartPt that) {
  int deltaX = this.x − that.x;
  int deltaY = this.y − that.y;
  CartPt p = new CartPt (deltaX, deltaY); // Abstandsvektor
  return p.distTo0();
}
```

Abstraktion zwischen Methoden derselben Klasse

- Wenn zwei Methoden ähnlich sind, . . .
  - Identifiziere das gemeinsame Muster.
  - Definiere neue Methode, die für die Unterschiede formale Parameter ansetzt.
  - Definiere die alten Methoden um, so dass sie die neue verwenden.

Beispiel: Simulation eines Lichtschalters

```
boolean draw() {
  if (this.on) {
    return this.paintOn();
  } else {
    return this.paintOff();
  }
}
boolean paintOn() {
  return this.c.drawRect(this.origin, this.width, this.height, this.light)
  && this.c.drawDisk(this.center, this.radius, this.dark);
}
boolean paintOff() {
  return this.c.drawRect(this.origin, this.width, this.height, this.dark)
  && this.c.drawDisk(this.center, this.radius, this.light);
}
```

Beispiel: Simulation eines Lichtschalters

```
boolean draw() {
  if (this.on) {
    return this.paintOn();
  } else {
    return this.paintOff();
  }
}
boolean paintOn() {
  return this.c.drawRect(this.origin, this.width, this.height, this.light)
  && this.c.drawDisk(this.center, this.radius, this.dark);
}
boolean paintOff() {
  return this.c.drawRect(this.origin, this.width, this.height, this.dark)
  && this.c.drawDisk(this.center, this.radius, this.light);
}
```

- Einziger Unterschied zwischen `paintOn()` und `paintOff()`: Vorder- und Hintergrundfarbe vertauscht.

Verallgemeinerte `paint()`-Methode

```
boolean paint(IColor front, IColor back) {
  return this.c.drawRect(this.origin, this.width, this.height, back)
  && this.c.drawDisk(this.center, this.radius, front);
}
boolean paintOn() {
  return paint (this.dark, this.light);
}
boolean paintOff() {
  return paint (this.light, this.dark);
}
```

Zusammenfassung

Abstraktion mit Vereinigungen von Klassen

Vergleich Nach dem Entwurf einer Vereinigung IUnion von Klassen, suche nach identischen Feld- und Methodendefinitionen. Felder müssen in *allen Varianten* auftreten. Methoden müssen in *mindestens zwei Varianten* auftreten.

Abstraktion Definiere die abstrakte Klassen AClass, die das Vereinigungsinterface mit abstrakten Methoden implementiert. In den Variantenklassen muss **implements** IUnion durch **extends** AClass ersetzt werden.
Eliminiere die gemeinsamen Felddefinitionen aus den Varianten und verschiebe sie in AClass. Ändere die Konstruktoren, sodass sie **super** für die verschobenen Felder verwenden.
Kopiere die gemeinsamen Methodendefinitionen nach AClass und eliminiere die Übereinstimmenden aus den Variantenklassen.

Zusammenfassung

Abstraktion mit Vereinigungen von Klassen/2

Superaufruf Falls eine Methode in einer Variantenklasse nicht gleich wie in AClass implementiert ist, versuche sie mit Hilfe eines Superaufrufs umzuschreiben. (Nicht zwingend erforderlich.)

Test Teste alle vorhandenen Beispiele. Sie sollten ohne Änderungen korrekt funktionieren.

Zusammenfassung

Abstraktion durch Vereinigung von Klassen

Vergleich Suche nach zwei oder mehr ähnlichen Klassen ohne Superklassen. Die Klassen sollten zumindest verwandte Funktionalität besitzen.

Falls möglich, können vorher Felder und Methoden umbenannt werden.

Abstraktion Definiere eine Vereinigung bestehend aus einem Interface IUnion und einer gemeinsamen Superklasse SClass.

Das Interface IUnion enthält die den Klassen gemeinsamen Methodenspezifikationen. (Nicht zwingend erforderlich.)

Die Superklasse SClass enthält die gemeinsamen Felder und Methodendefinitionen der ursprünglichen Klassen. Formuliere eine verallgemeinerte Zweckangabe für SClass.

Wandle die Klassen in Subklassen von SClass um (entferne überflüssige Feld- und Methodendefinitionen, ändere die Konstruktoren).

Testen Führe die Testumgebungen für die ursprünglichen Klassen aus. Falls SClass selbst sinnvolle Anwendungen hat, stelle hierfür eine Testumgebung zusammen.

Zusammenfassung

Subklassen von Bibliotheksklassen

- Beim Entwurf einer Klassenhierarchie kann sich herausstellen, dass Klassen mit der gesuchten Funktionalität bereits existieren.
  - Es gibt bereits eine Klasse mit einem allgemeineren Zweck, die als Superklasse einer Vereinigung dient. Dann kann die neue Klasse eine neue Variante werden.
  - Es gibt eine Bibliothek, die eine ggf. passende Superklasse bereitsstellt. Es ist sehr vorteilhaft, die neue Klasse zu einer Subklasse zu machen, da so die Infrastruktur der Bibliothek (Methodendefinitionen usw) verwendet werden kann.
    Viele Bibliotheken definieren zu diesem Zweck abstrakte Klassen.
    Nachteil: Die Methoden und Felder einer Bibliotheksklasse können nicht verändert werden.
- Zu beachten ist nun:
  - Welche Felder sind vorhanden, welche müssen hinzugefügt werden?
  - Welche Methoden der Superklasse besitzen die richtige Funktionalität?
  - Welche Methoden der Superklasse müssen überschrieben (und/oder erweitert) werden?
  - Welche Methoden müssen von Grund auf entwickelt werden?

Epilog

Smarte Konstruktoren

```
// zur Erinnerung:
class Date {
  int day; // zwischen 1 und 31
  int month; // zwischen 1 und 12
  int year; // größer oder gleich 1900

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```

Epilog

Smarte Konstruktoren

```
// zur Erinnerung:
class Date {
  int day; // zwischen 1 und 31
  int month; // zwischen 1 und 12
  int year; // größer oder gleich 1900

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```

- Nicht jedes Datum, das der Konstruktor erzeugen kann, ist sinnvoll:

```
new Date (−1, 77, 333)
```

Epilog

Smarte Konstruktoren

```
// zur Erinnerung:
class Date {
  int day; // zwischen 1 und 31
  int month; // zwischen 1 und 12
  int year; // größer oder gleich 1900

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```

- Nicht jedes Datum, das der Konstruktor erzeugen kann, ist sinnvoll:

```
new Date (−1, 77, 333)
```

- Aber der Konstruktor kann sinnlose Daten verhindern, indem er eine Prüfung zwischenschaltet und ggf. eine Exception auslöst.

Smarter Konstruktor für Date

```
// Datum, bei dem der Konstruktor die meisten falschen Eingaben ausschließt
class Date {
  int day; // zwischen 1 und 31
  int month; // zwischen 1 und 12
  int year; // größer oder gleich 1900

  Date(int day, int month, int year) {
    if (1 <= day && day <= 31
      && 1 <= month && month <= 12
      && 1900 <= year) {
      this.day = day;
      this.month = month;
      this.year = year;
    } else {
      InputOutput.error("the given numbers do not specify a date");
    }
  }
}
```

Smarter Konstruktor mit Hilfsmethode

```
class Date {
  int day; // zwischen 1 und 31
  int month; // zwischen 1 und 12
  int year; // größer oder gleich 1900

  Date(int day, int month, int year) {
    if (this.valid (day, month, year)) {
      this.day = day;
      this.month = month;
      this.year = year;
    } else {
      InputOutput.error("the given numbers do not specify a date");
    }
  }
  // in Java wäre ''valid'' eine static Methode: sie benutzt die Felder nicht
  boolean valid (int day, int month, int year) {
    return 1 <= day && day <= 31
        && 1 <= month && month <= 12
        && 1900 <= year;
  }
}
```

Konstruktor mit Initialisierung

Betrachte ein Spiel, bei dem am oberen Rand des Bildschirms auf Position (10,20) jeweils ein Block erscheint, der mit konstanter Geschwindigkeit fällt. Der Block bleibt auf dem ersten Hindernis liegen, dann erscheint ein neuer Block. Der Spieler kann den Block nach rechts und links bewegen und muss versuchen, dass kein Blockstapel bis zum oberen Rand wächst.

- Frage: Wie wird ein "fallender" Block modelliert?
- Klar: Die Position des Blocks wird mit einem Paar von Koordinaten (x,y) beschrieben.
- Wunsch:
  - Die Startposition wird im Konstruktor gesetzt.
  - Bei nachfolgenden Positionen ändern sich die (x,y) Koordinaten.

Fallender Block

Versuch: Initialisiere Startposition im Rumpf der Klasse

```
class DrpBlock {
  int x = 10;
  int y = 20;
  DrpBlock () { }
  ...
}
```

- Passt nicht, da sich `x` und `y` nicht ändern können.
- Generell ausgeschlossen.

Fallender Block

Versuch: Initialisierung im Konstruktor

```
class DrpBlock {
  int x;
  int y;
  DrpBlock () {
    this.x = 10;
    this.y = 20;
  }
  ...
}
```

- Keine weitere Änderung möglich.
- Führt aber zur gewünschten Lösung

Überladung von Konstruktoren

Konstruktoren können *überladen* werden.

- Es dürfen **mehrere Konstruktoren** für dieselbe Klasse definiert werden.
- **Die Konstruktoren müssen sich in Anzahl oder Typ der Argumente unterscheiden**, damit Java für jeden Aufruf entscheiden kann, welcher Konstruktor gemeint ist.

Überladung von Konstruktoren

Fallender Block

```
class DrpBlock {
    int x;
    int y;
    int SIZE = 10;
    // Konstruktor für Initialisierung
    DrpBlock () {
        this.x = 10;
        this.y = 20;
    }
    // Konstruktor für fallende Blöcke (Aufruf nur innerhalb von DrpBlock)
    DrpBlock (int x, int y) {
        this.x = x;
        this.y = y;
    }

    DrpBlock drop() {
        return new DrpBlock (this.x, this.y + 1);
    }

    public void draw (Canvas c) {
```

Kapselung und Geheimhaltung

- *Kapselung* bedeutet das Verstecken bzw. Geheimhalten von Feldern, Konstruktoren und Methoden einer Klasse gegenüber Objekten einer anderen Klasse.
- Idee
  - Innerhalb der Klasse gibt es keine Zugriffsbeschränkungen.
  - Außerhalb der Klasse sollen Anwender nur die Verträge und Signaturen von bekannten Feldern, Konstruktoren und Methoden verwenden, damit sie keine internen Konventionen (*Invarianten*) verletzen.
- In Programmiersprachen wird Kapselung durch Geheimhaltungsattribute spezifiziert.

Geheimhaltung in Java

- Die Geheimhaltungsattribute können der Definition von Feldern, Konstruktoren und Methoden vorangestellt werden.
- Die folgenden Attribute sind verfügbar.

  private Das Subjekt darf **nur innerhalb der definierenden Klasse** verwendet werden. (Das geht immer.)

  public Das Subjekt kann **von allen Klassen verwendet werden**.

  protected Das Subjekt darf **nur in Subklassen der definierenden Klasse** verwendet werden. (Auch in deren Subklassen usw.)

  (ohne Attribut) Das Subjekt darf nur von Klassen im gleichen Paket verwendet werden.
- In einer Subklasse dürfen keine Felder, Konstruktoren oder Methoden "verschwinden".

Geheimhaltung in Java

Beispiel: Fallender Block

```
class DrpBlock {
    private int x;
    private int y;
    private int SIZE = 10;
    // Konstruktor für Initialisierung
    public DrpBlock () {
        this.x = 10;
        this.y = 20;
    }
    // Konstruktor für fallende Blöcke (Aufruf nur innerhalb von DrpBlock)
    private DrpBlock (int x, int y) {
        this.x = x;
        this.y = y;
    }

    public DrpBlock drop() {
        return new DrpBlock (this.x, this.y + 1);
    }
}
```

Welche Geheimhaltungsstufe?

Richtlinien

- Im Zweifelsfall immer **private**.
- Eine Interfacemethode muss **public** sein.
- Eine Methode, die von einer anderen Klasse verwendet werden muss: **public**.
- Ein Feld oder Methode, die nur von Subklassen benutzt werden sollen: **protected**.
- Ein Feld, das global bekannt sein muss oder von einer anderen Klasse verwendet werden soll: **public**.
  Sehr selten notwendig. Vermeiden!
- Ein Konstruktor einer abstrakten Klasse soll nur von den Subklassen verwendet werden: **protected**.
- Ein Konstruktor, der nur von der Klasse selbst verwendet werden darf: **private**.