Klassen und ihre Beziehungen: Mehrfache Vererbung, Schnittstellen und Pakete

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

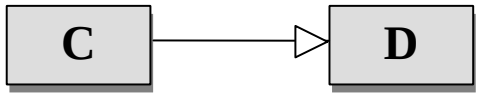
Wichtig: Klausuranmeldung

- **1. Teilklausur am 16.6.2006 !**
- **Klausuranmeldung ist für Klausurteilnahme erforderlich!**
- Die Anmeldung zur Klausur ist jetzt über UniWorx möglich.
- Bitte bis zum 13.6. anmelden.

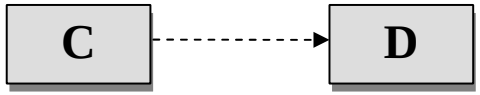
Ziele

- Den Begriff der einfachen und mehrfachen Vererbung verstehen
- Schnittstellendeklarationen kennen lernen
- Pakete zur Strukturierung von Programmsystemen kennen lernen

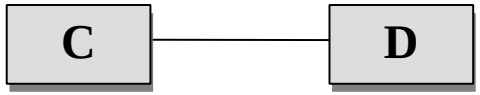
Beziehungen zwischen Klassen



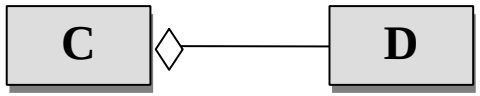
Vererbung: Die Klasse C ist Erbe der Klasse D



Abhängigkeit: Die Klasse C benützt Elemente der Klasse D (i.a. Methoden)



Assoziation: Die Klassen C und D stehen in Beziehung

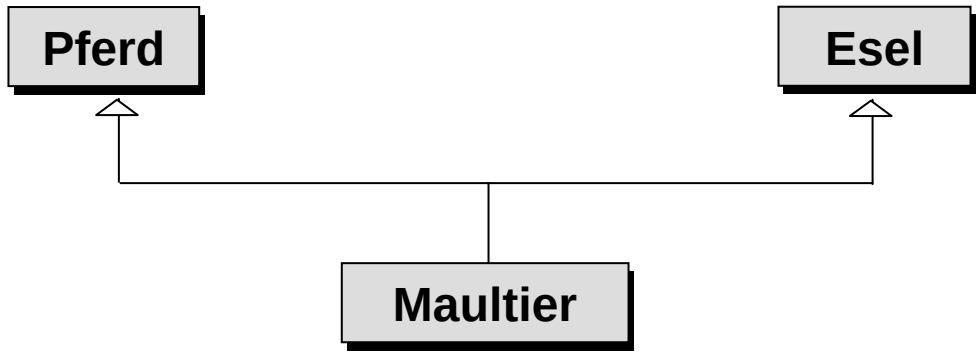


(schwache) Aggregation: Jedes Objekt von C enthält Objekte von D

Einfache und mehrfache Vererbung

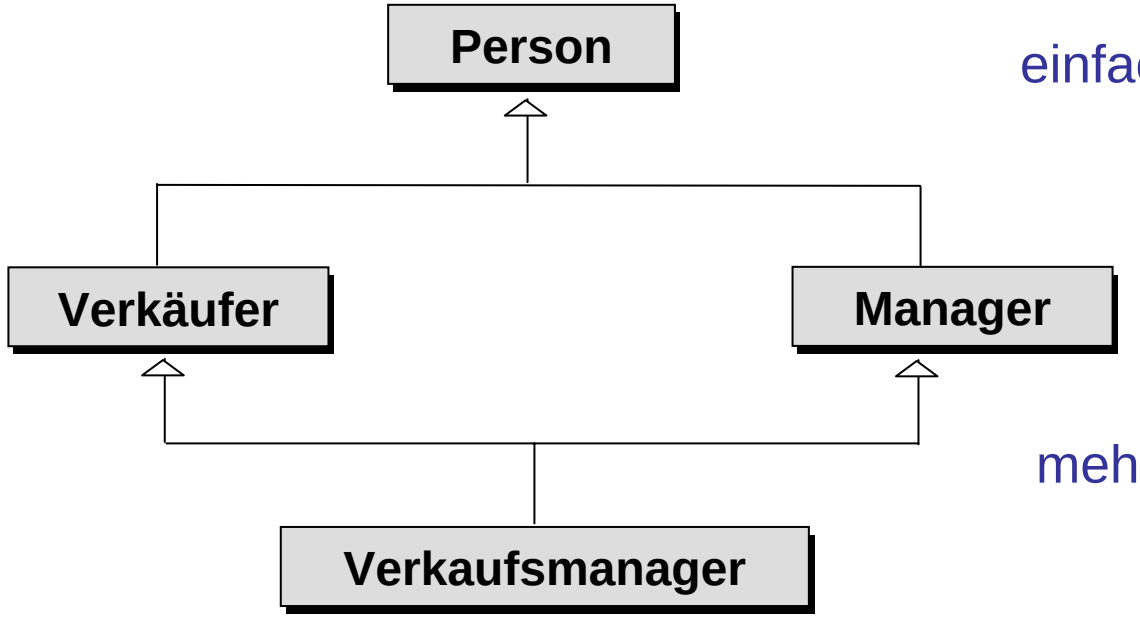
- Man spricht von **einfacher Vererbung**, wenn jede Klasse höchstens eine direkte Superklasse besitzt.
- Kann eine Klasse mehrere direkte Superklassen haben, spricht man von **mehrfacher Vererbung.**

Beispiel: mehrfache Vererbung



Einfache und mehrfache Vererbung

Beispiel: einfache und mehrfache Vererbung



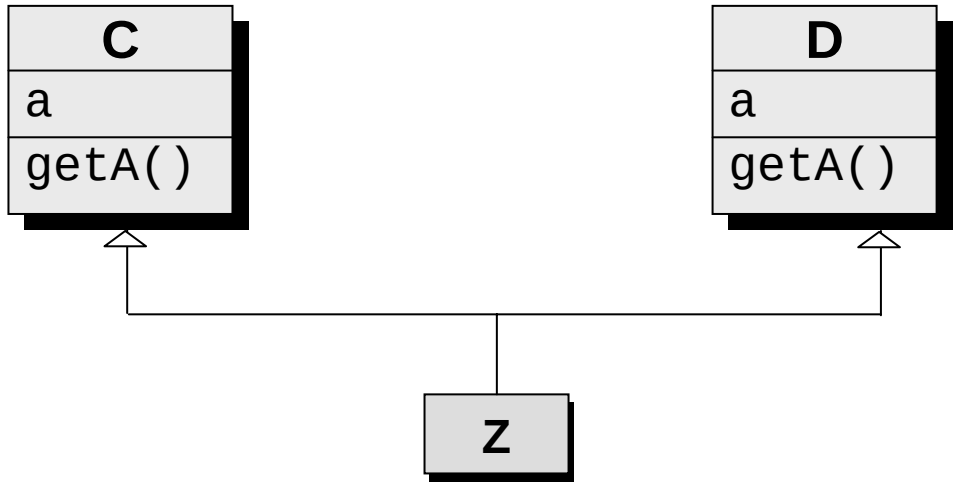
einfache Vererbung

mehrfache Vererbung

Bemerkung: Java unterstützt bei **Klassen** nur **einfache Vererbung.**

Einfache und mehrfache Vererbung

Problem bei mehrfacher Vererbung: mögliche Mehrdeutigkeiten beim Verwenden von Attributen und Methoden:



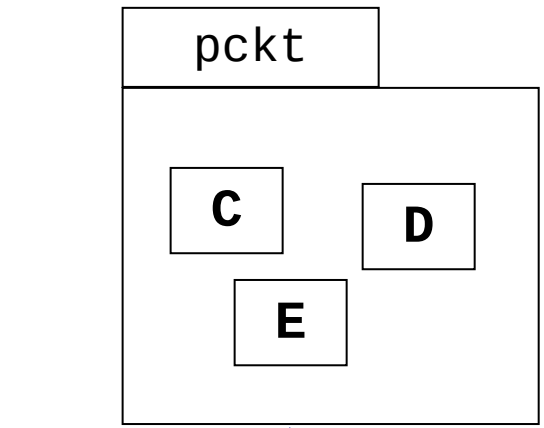
Welches Attribut a und welche Methode `get(A)` sollte Z erben und, wenn jeweils beide geerbt werden, welche der Methoden wird beim Aufruf von `get(A)` ausgewählt?

Pakete

- Pakete dienen zur Strukturierung großer Programmsysteme.
  Ein Paket (Schlüsselwort „**`package`**“) fasst verwandte Klassen zusammen.
- Ein Paket hat einen Namen, der mit einem kleinen Buchstaben beginnt, und steht in einem Verzeichnis mit dem gleichen Namen.
- Alle Klassen eines Pakets `pckt` stehen in demselben Verzeichnis, aber in unterschiedlichen Dateien. **Jede** dieser Dateien enthält die Anweisung

  ```
  package pckt;
  ```

  als erste Anweisung.



Darstellung in UML: Paket `pckt` mit 3 Klassen

Beispiel

Datei: point\Point.java

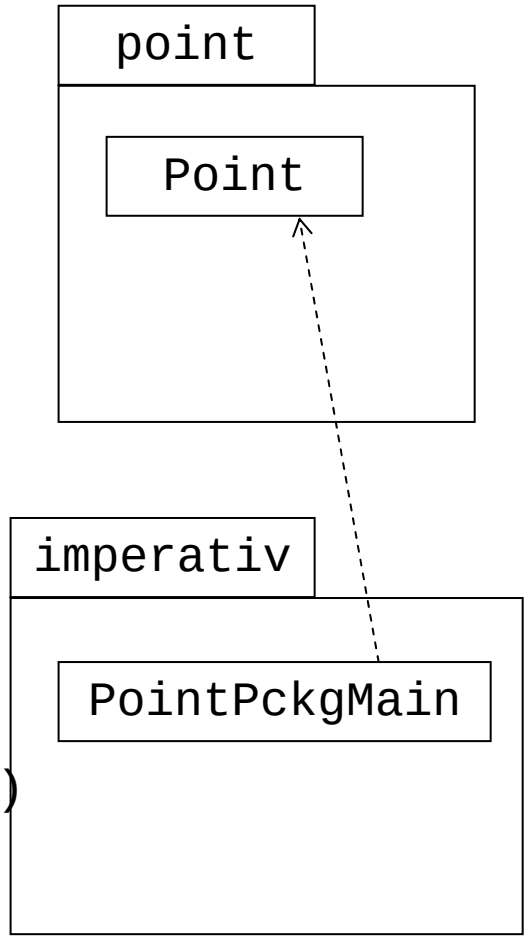
```
package point;
public class Point
{  private int x, y;
   public Point(int a, int b)
   { ...}
}
```

Klasse im Paket point

Datei: imperativ\PointPckgMain.java

```
package imperativ;
import point.Point;
public class PointPckgMain
{ public static void main(String[] args)
  {    Point p = new Point(10,20); ....
  }
}
```

Benutzung einer Klasse von point



Benutzung von Paketen

- **Eine einzelne** Klasse C eines Pakets `pckt` kann in einem anderen Paket benutzt werden durch die Anweisung

```
import pckt.C;.
```

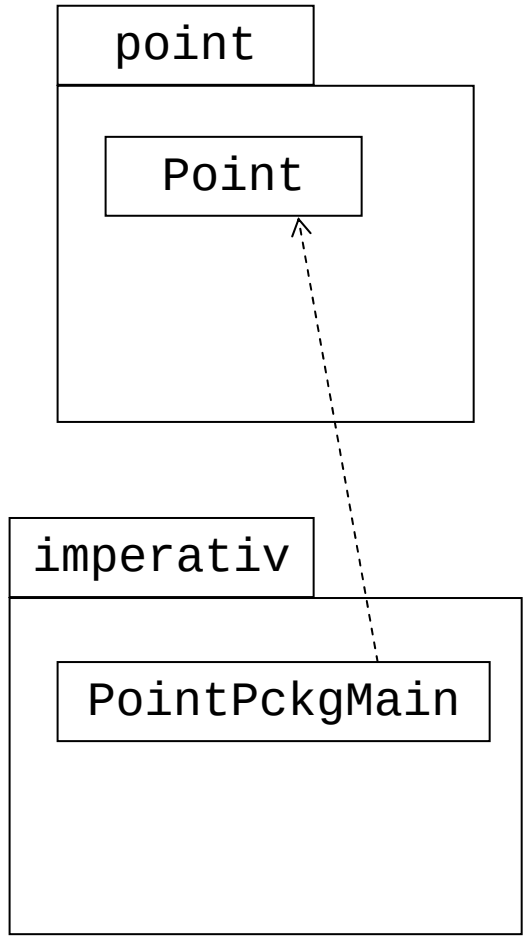
- **Alle Klassen** eines Pakets `pckt` werden benutzt durch

```
import pckt.*;.
```

- Klassen **unterschiedlicher Pakete** `pckt1,..., pckt3` werden benutzt durch

```
import pckt1.C;
import pckt2.*;.
import pckt3.D;. ...          .
```

Benutzung von Paketen: Übersetzung und Ausführung von Programmen



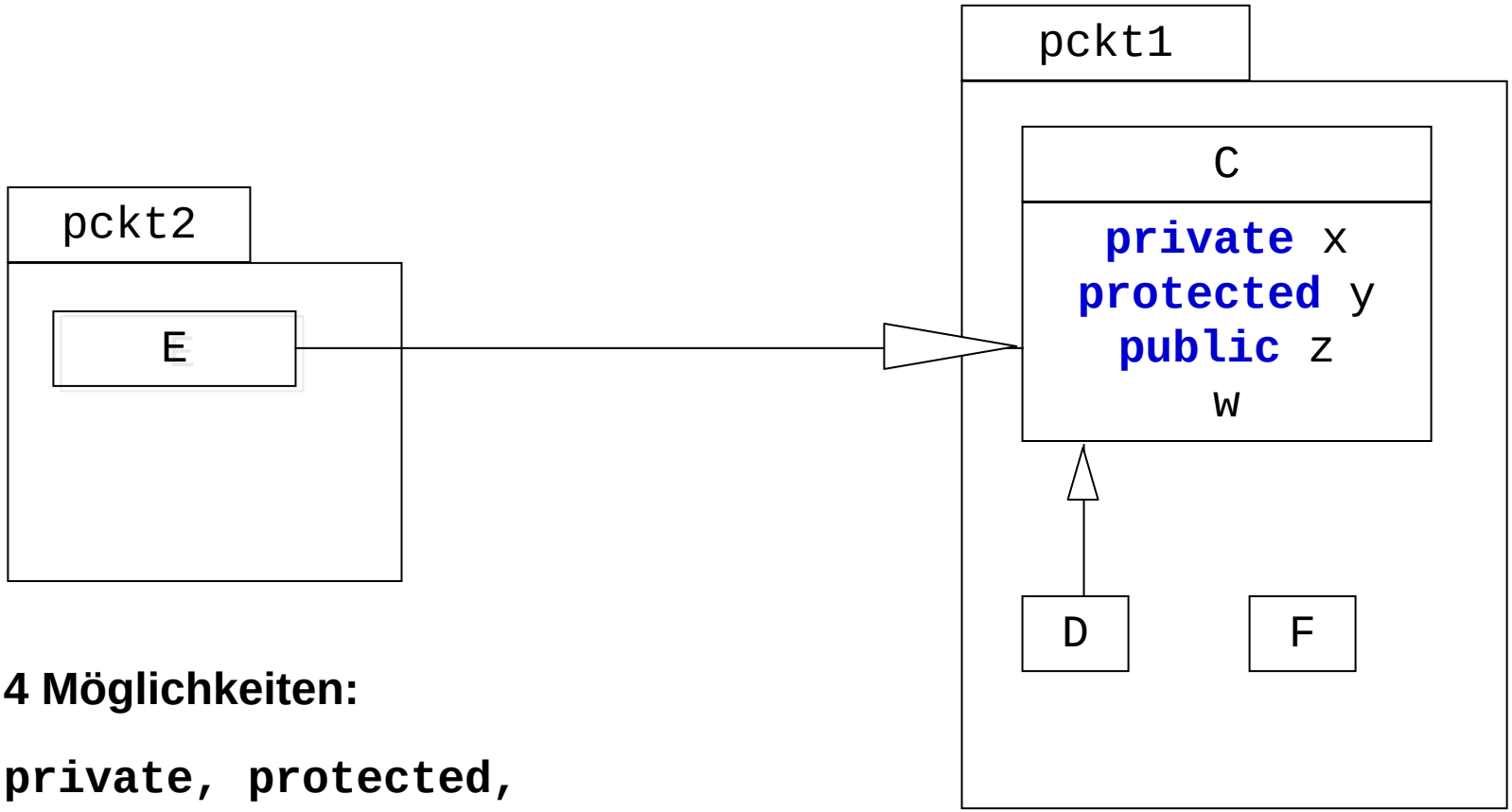
- **Wähle ein Basisverzeichnis** (z.B. mit Namen java0) mit den Unterverzeichnissen `point` und `imperativ`, in denen alle Dateien jeweils als erste Anweisung „`package point`“ bzw. „`package imperativ`“ enthalten.
- **Wechsle** in das Verzeichnis „java0“.
- **Übersetze** die **Datei** `PointPackgMain.java` mittels

  java0> `javac imperativ\PointPackgMain.java (Windows)`

  Oder **java0>** `javac imperativ/PointPackgMain.java (Unix)`
- **Führe** die **Klasse** `PointPackgMain` aus mittels

  java0> `java imperativ.PointPackgMain` .

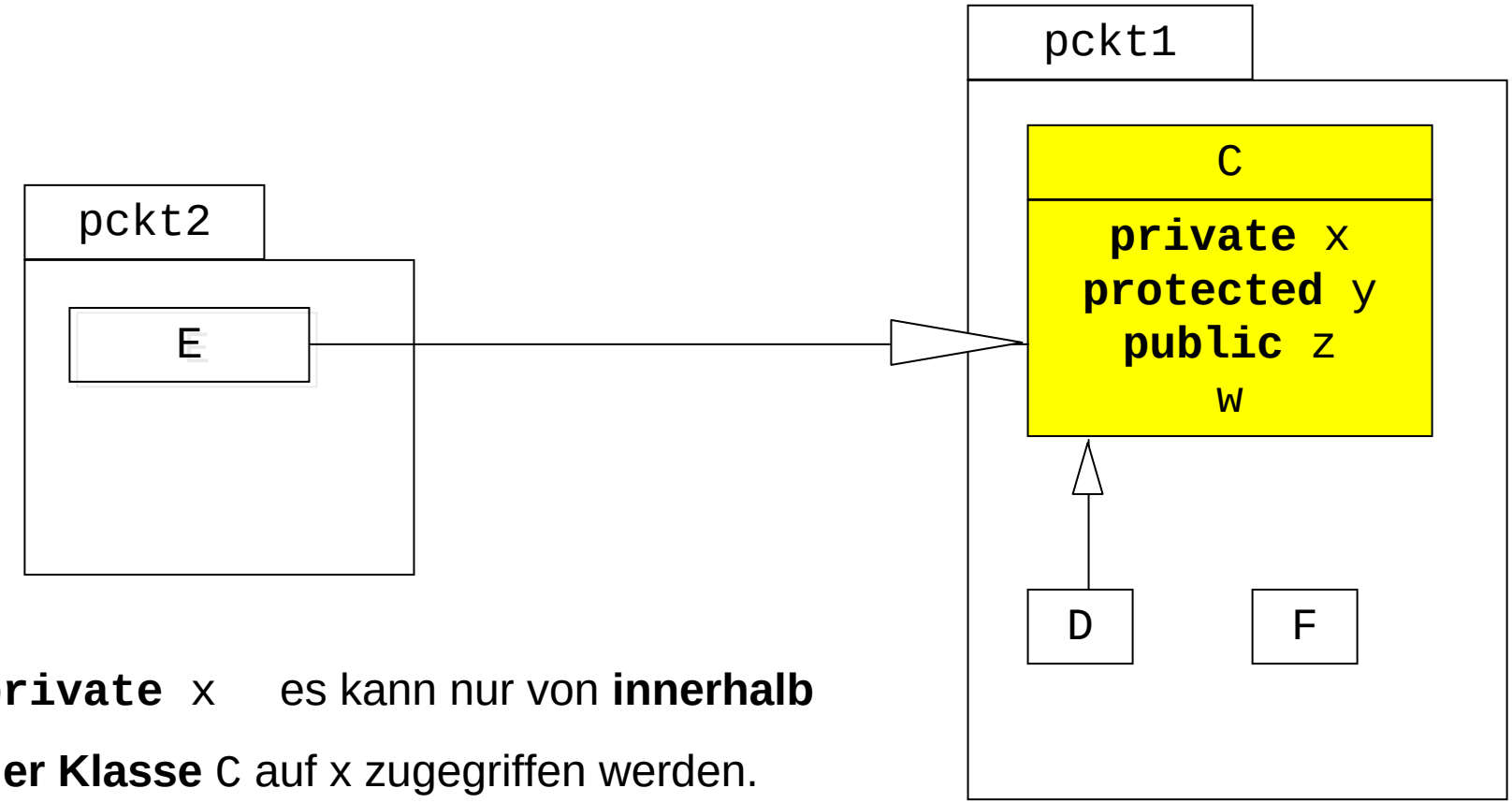
Sichtbarkeitsregeln für Identifikatoren



- **4 Möglichkeiten:**

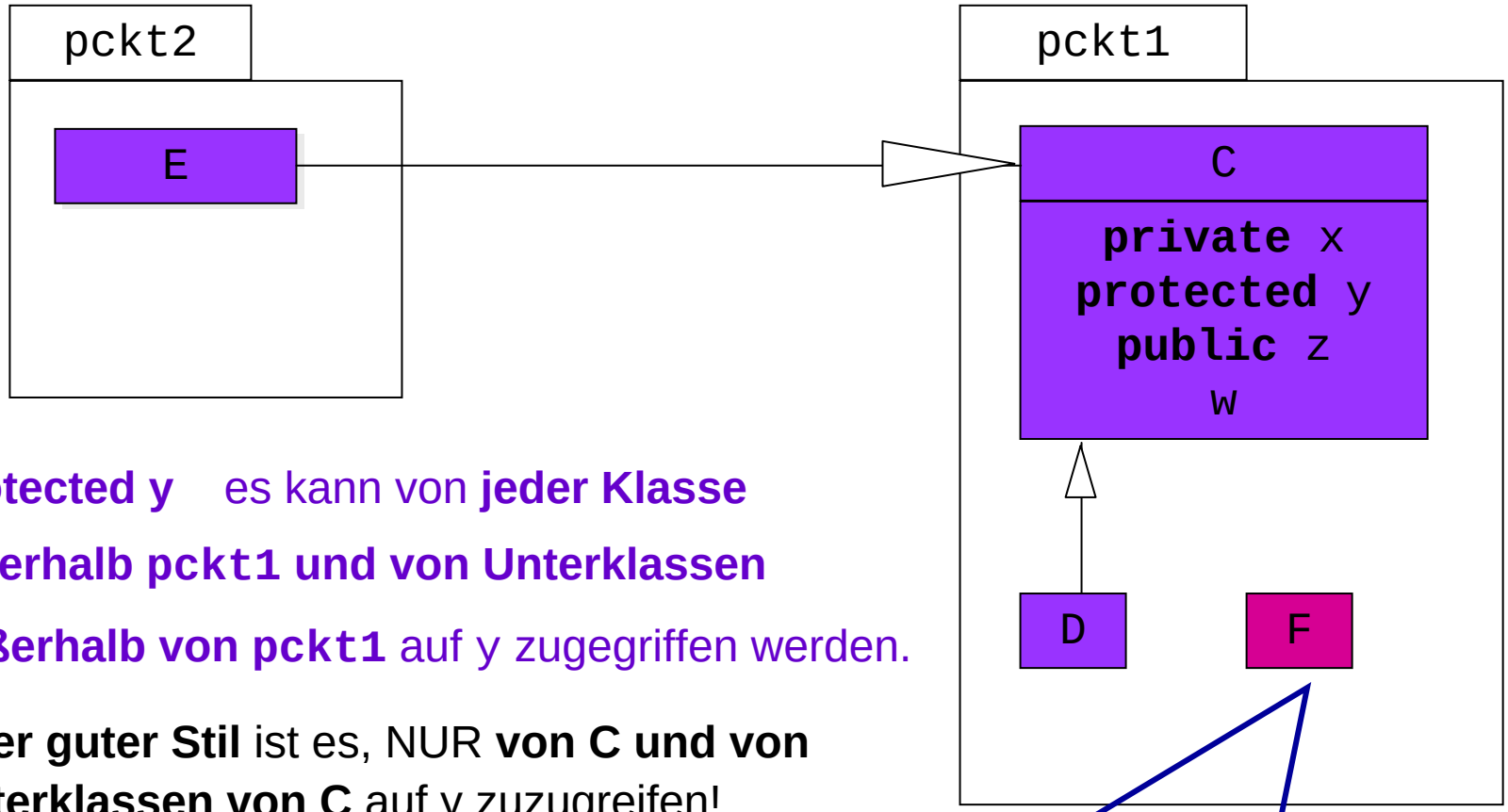
  `private, protected,`
  `public, kein Modifier`

Sichtbarkeitsregeln für Identifikatoren : private



- **`private` x** es kann nur von **innerhalb der Klasse** `C` auf x zugegriffen werden.

Sichtbarkeitsregeln für Identifikatoren : protected

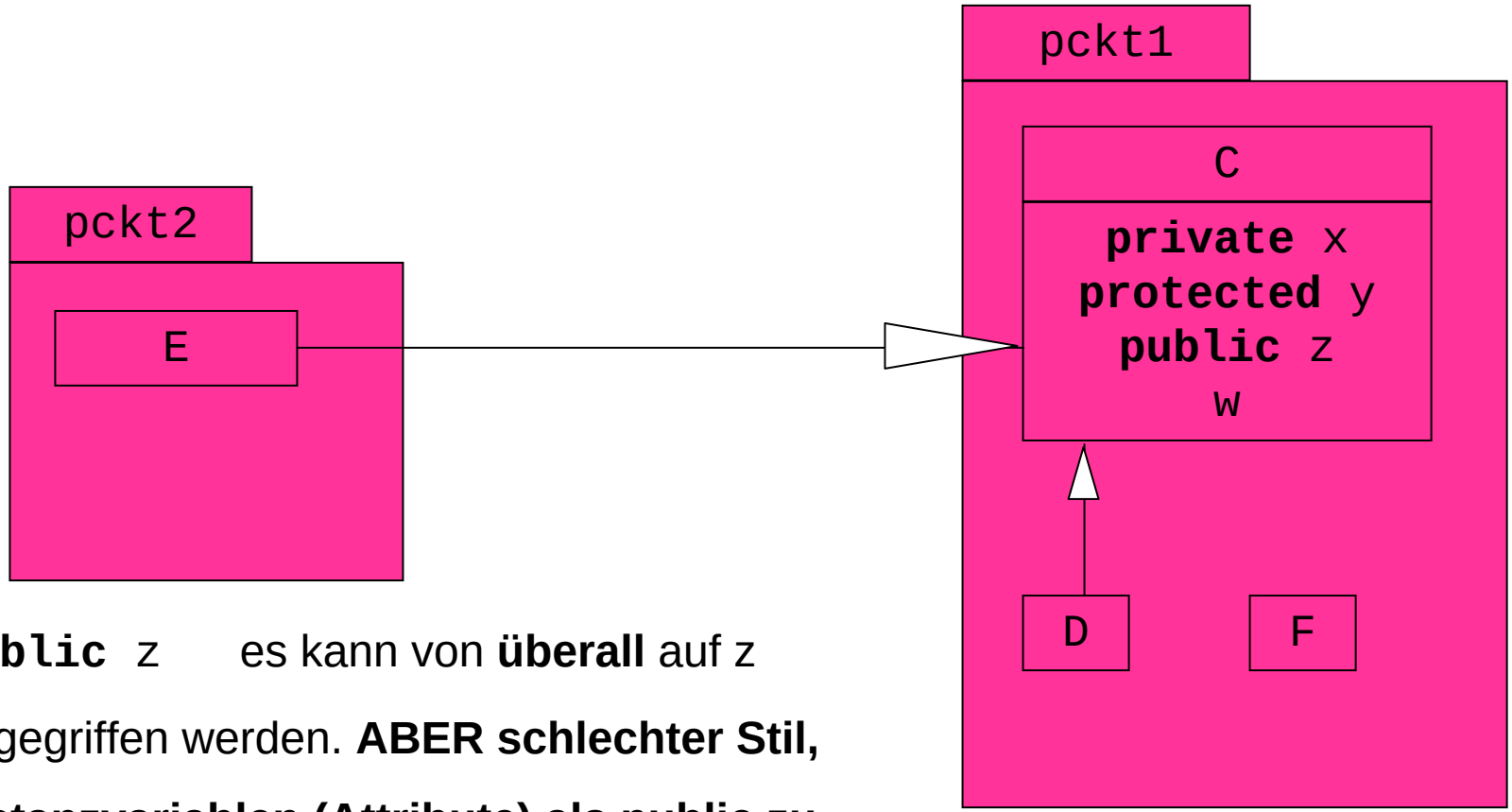


- **protected y** es kann von **jeder Klasse innerhalb `pckt1` und von Unterklassen außerhalb von `pckt1`** auf y zugegriffen werden.

Aber guter Stil ist es, NUR **von C und von Unterklassen von C** auf y zuzugreifen!

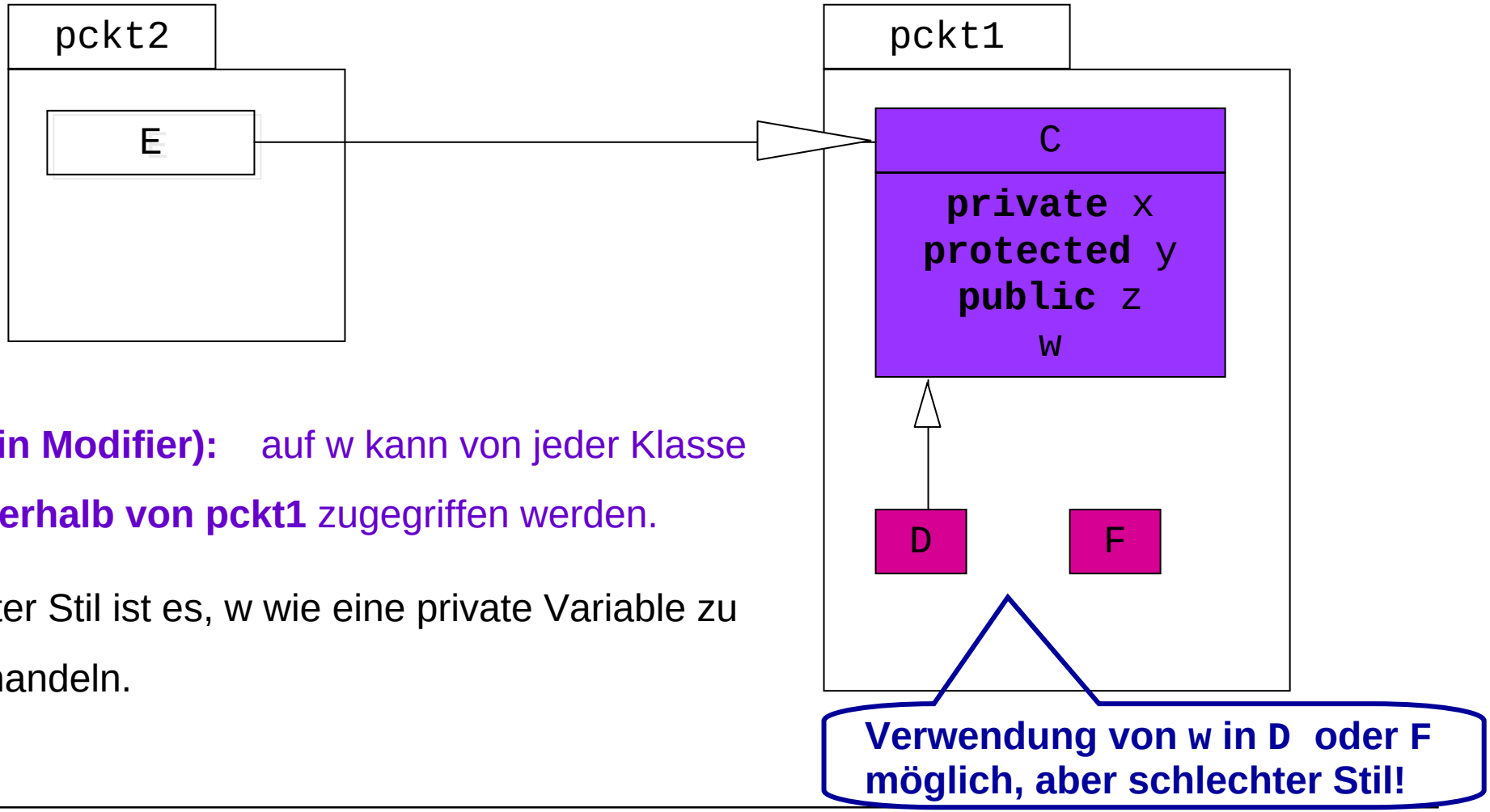
protected kann bei Vererbung **private** ersetzen.

Sichtbarkeitsregeln für Identifikatoren : public



- **`public`** z es kann von **überall** auf z zugegriffen werden. **ABER schlechter Stil, Instanzvariablen (Attribute) als public zu deklarieren!**

Sichtbarkeitsregeln für Identifikatoren: kein Modifier



- **(kein Modifier):** auf w kann von jeder Klasse **innerhalb von pckt1** zugegriffen werden.

Guter Stil ist es, w wie eine private Variable zu behandeln.

Sichtbarkeitsregeln für Identifikatoren

Die folgenden Regeln gelten für alle Identifikatoren: Lokale Variablen, Methoden, Klassen,...

- **public** es kann von **überall** darauf zugegriffen werden. Aber beachte Stil!
- **protected** es kann von jeder Klasse **innerhalb dieses Pakets und von Unterklassen außerhalb des Pakets** zugegriffen werden. Aber beachte Stil!
- **private** es kann nur von **innerhalb der Klasse** zugegriffen werden.
- **(keine Beschreibung)** es kann von jeder Klasse **innerhalb dieses Pakets** zugegriffen werden. Aber beachte Stil!

Wiederverwendbare Lösungen?

- Wie kann man eine Klasse, die Dienste für einen speziellen Datentyp anbietet, für andere Datentypen wiederverwenden?
- **Beispiel:** Die Klasse `NumberSet` berechne das Maximum und den Durchschnitt einer Menge von Zahlen.
  - Wie kann man `NumberSet` verallgemeinern, um für eine Menge von Bankkonten den maximalen und den durchschnittlichen Kontostand berechnen?
  - Oder wie kann man für eine Menge von Punkten den größten Punkt und den Durchschnitt berechnen? .

Beispiel NumberSet

```
public class NumberSet
{       private double sum;
        private double maximum;
        private int count;

        public NumberSet()
        {       sum = 0; count = 0;  maximum = 0;  }

        public void add(double x)
        {       sum = sum + x;
                if (count == 0 || maximum < x) maximum = x;
                count++;
                }

        public double getMaximum()
        {       return maximum;
        }
}
```

Konstruiert leere Zahlenmenge

Fügt x zu der Zahlenmenge hinzu

Berechnet Maximum

Versuch: Modifikation für BankAccount-Objekte

```
public class AccountSet          //Modifikation von NumberSet für BankAccount Objekte
{       private double sum;
        private BankAccount maximum;
        private int count;

        ...

        public void add(BankAccount x)
        {       sum = sum + x.getBalance() ;
                if (count == 0
                        || maximum.getBalance() < x.getBalance() )
                    maximum = x;
                count++;
        }

        public BankAccount getMaximum()
        {       return maximum;
        }
}
```

Enge Kopplung von BankAccount und den AccountSet-Operationen!

Fügt x zu den Bankkonten hinzu

Berechnet Maximum

Versuch: Modifikation für Point-Objekte

```
public class PointSet     //Modifikation von NumberSet für Point Objekte
{       private double sum;
        private Point maximum;
        private int count;

        ...

        public void add(Point p)
        {       sum = sum + p.getDistFromZero();
                if (count == 0
                        || maximum.getDistFromZero() < p.getDistFromZero()
   )
                    maximum = p;
                count++;
        }

        public Point getMaximum()
        {       return maximum;
        }
```

Enge Kopplung von Point und den PointSet-Operationen!

Berechnet Abstand $\sqrt{x^2+y^2}$ vom Ursprung

Berechnet Maximum

Besser: Verwende Schnittstelle

```
public interface Measurable
{       double getMeasure();
}
```

Allgemeine Schnittstelle für messbare Objekte

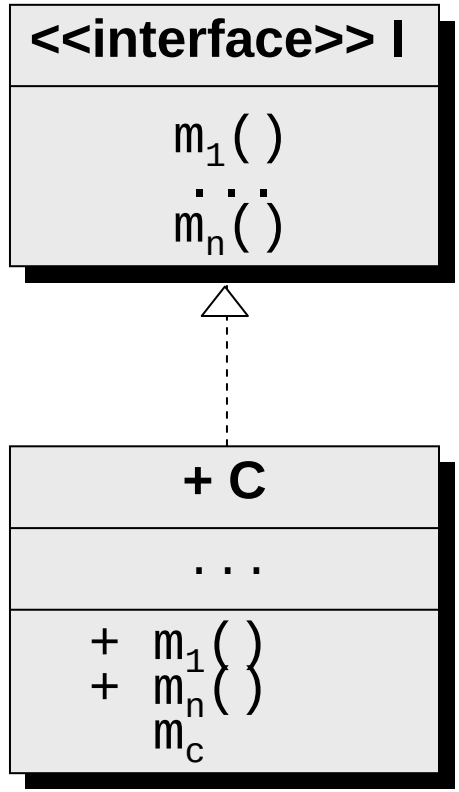
```
public class DataSet
{       private double sum;
        private Measurable maximum;
        private int count;

        ...

        public void add(Measurable p)
        {       sum = sum + p.getMeasure() ;
                ...
        }
```

Allgemeine Klasse, die `Measurable` verwendet, aber NICHT die Implementierungen von Measurable kennt

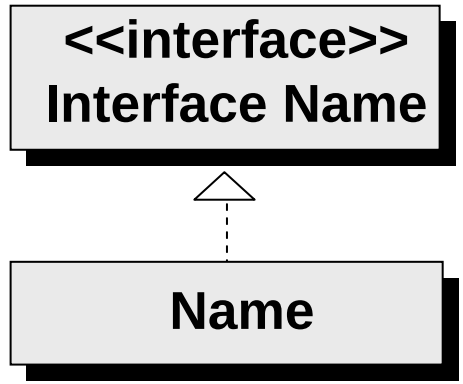
Lose Kopplung mit Implementierung erlaubt Wiederverwendung!

Schnittstellen

- Eine **Schnittstelle** in Java (Schlüsselwort „**`interface`**“) deklariert eine **Menge von Methoden** (ohne Angabe eines Rumpfs) und **Konstanten** (**aber** keine Attribute). Man nennt eine Methode ohne Rumpf „abstrakte Methode“. Im Gegensatz zu Klassen ist Mehrfachvererbung erlaubt, d.h. eine Schnittstelle kann Erbe mehrerer Schnittstellen sein.
- Eine Klasse `C` **implementiert** eine Schnittstelle `I`, wenn **alle Methoden** der Schnittstelle in `C` mit ihrer exakten Funktionalität **implementiert werden**, und zwar durch „öffentliche“ Methoden.



Schnittstellen



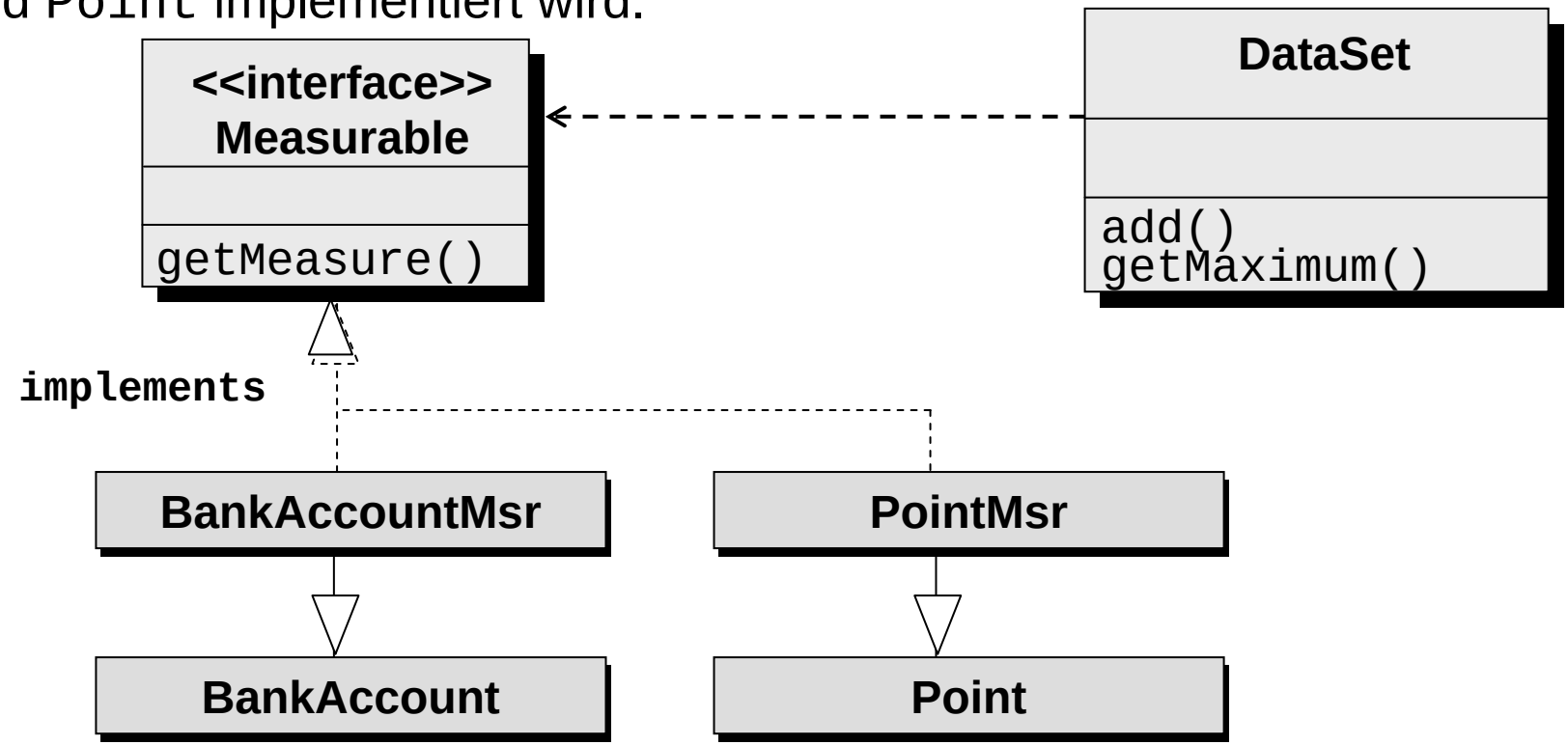
Schema:

```
public interface InterfaceName
  {
     Konstanten
     MethodenSignaturen // Methodenkoepfe fuer „public“ Methoden
  }

public class Name implements InterfaceName
  {
    private Attribute
    public  Methoden    // mindestens eine konkrete Methode fuer
                        // jede abstrakte Methode aus InterfaceName

  }
```

Schnittstellen

Beispiel: (1) Wiederverwendbare Lösung für das „Mengenproblem“

`DataSet` verwendet die Schnittstelle `Measure`, die von `BankAccount` und `Point` implementiert wird.



`getMeasure()` **muss** in `BankAccountMsr` und `PointMsr` implementiert werden

Implementierungen der Schnittstelle Measurable

```
public class BankAccountMsr extends BankAccount implements Measurable
{       public BankAccountMsr(double b)
        { super(b);
        }
        public double getMeasure()
        {       return getBalance();
        }
}

public class PointMsr extends Point implements Measurable
{       public PointMsr(int x, int y)
        { super(x,y);
        }
        public double getMeasure()
        {       return Math.sqrt(getX()*getX() + getY()*getY());
        }
}
```

Verwendung der Messoperationen

```
DataSet bankData = new DataSet();

bankData.add(new BankAccountMsr(0));
bankData.add(new BankAccountMsr(10000));

System.out.println("Average balance = " + bankData.getAverage());

Measurable max = bankData.getMaximum();
System.out.println("Highest balance = " + max.getMeasure());

max.getBalance();   //Fehler, da max vom statischen Typ Measurable!!
```

Problem: Nachträgliche enge Kopplung mit BankAccount kann zu vielen Erbenklassen führen

Man kann eine Bankkonto-Instanz durch „Typecast“ erhalten, wenn die Instanz den passenden Typ besitzt:

```
BankAccountMsr b = (BankAccountMsr)max;
```

Verwendung der Messoperationen (2)

- Wenn eine Klasse ein Interface implementiert, kann man ein Objekt dieser Klasse in ein Objekt der Schnittstelle konvertieren:

```
PointMsr p = new PointMsr(1,3);
max = p;                    //OK, da PointMsr Measurable implementiert
```

- Man kann aber nie eine Schnittstelle instanzieren:

```
max = new Measurable(); // Fehler!!
```

- Und einer Schnittstellenvariablen keine Instanz zuweisen, die die Schnittstelle **nicht** implementiert:

```
max = new Point(); //Fehler, da Point Measurable NICHT
                   // implementiert!!
```

Entwurfsvarianten zur Wiederverwendung mit Schnittstellen

3 Varianten:

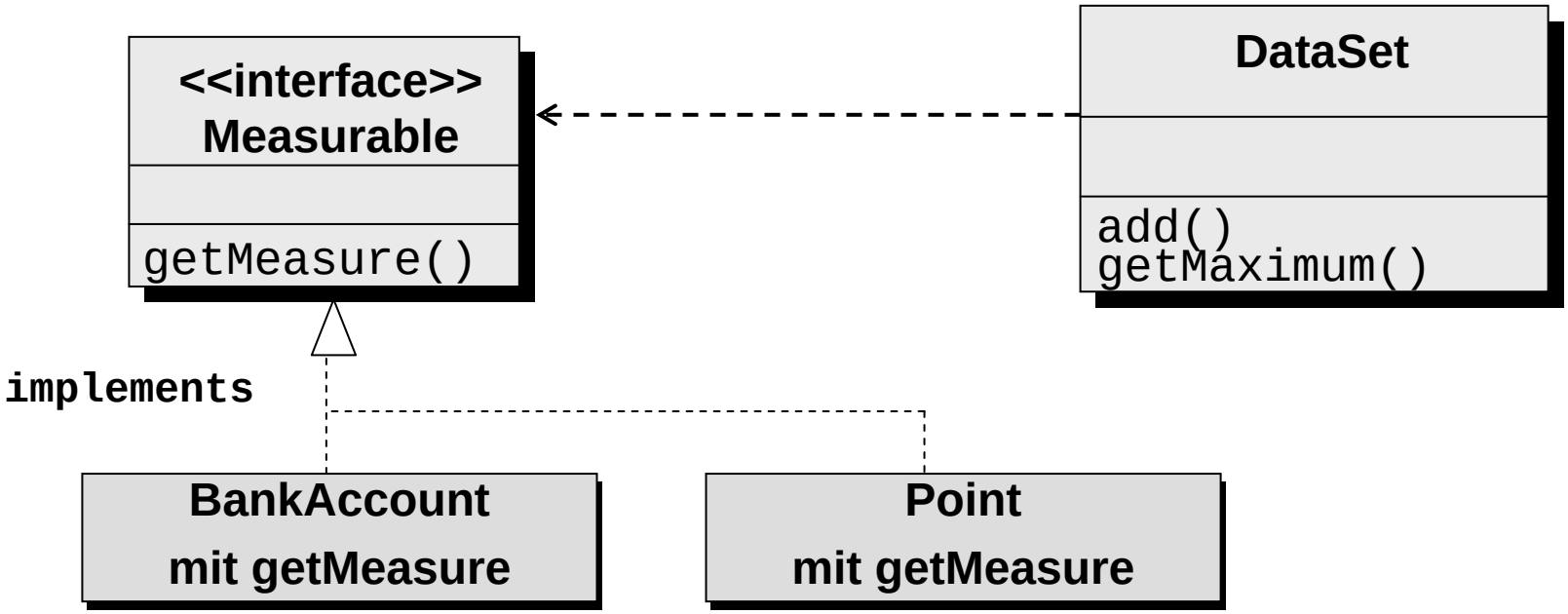
a) Verwendung der **`Measurable` Schnittstelle durch Erbenklassen** (`BankAccountMsr, PointMsr`, ...) der „gemessenen“ Klassen: Gut, wenn pro Klasse **nur eine Art von Messung** gebraucht wird und die „**gemessenen“ Klassen nicht verändert** werden sollen (siehe vorhergehende Folien).

b) Verwendung von **`Measurable`**, aber **direkte Änderung des Quellcodes der „gemessenen“ Klasse** durch Hinzufügen der Operation `getMeasure(): Gut` zur **Entwurfszeit** der gemessenen Klasse und wenn nur **eine Art von Messung** gebraucht wird; nicht möglich bei Klassen der Java API, die nicht verändert werden können.

c) (Strategiemuster) Verwendung einer Schnittstelle **`Measurer`** mit Operation `getMeasure(Object anObject)`, die das **gemessene Objekt als Parameter** übergibt. Gut, wenn mehrere Eigenschaften zu messen sind (z.B. Größe der x-Koordinate, Größe der y-Koordinate,... bei `Point`).

(b) Schnittstelle für eine Messoperation

Wiederverwendbare Lösung für das „Mengenproblem“
durch **direkte Änderung des Quellcodes der „gemessenen“ Klasse:**



`getMeasure()` **wird** in `BankAccount` und `Point` ergänzt

(b) Ergänzung der Implementierungsklassen

```
public class BankAccount implements Measurable
{       ... <alte BankAccount-Implementierung>
        public double getMeasure()
        {       return getBalance();
        }
}
```

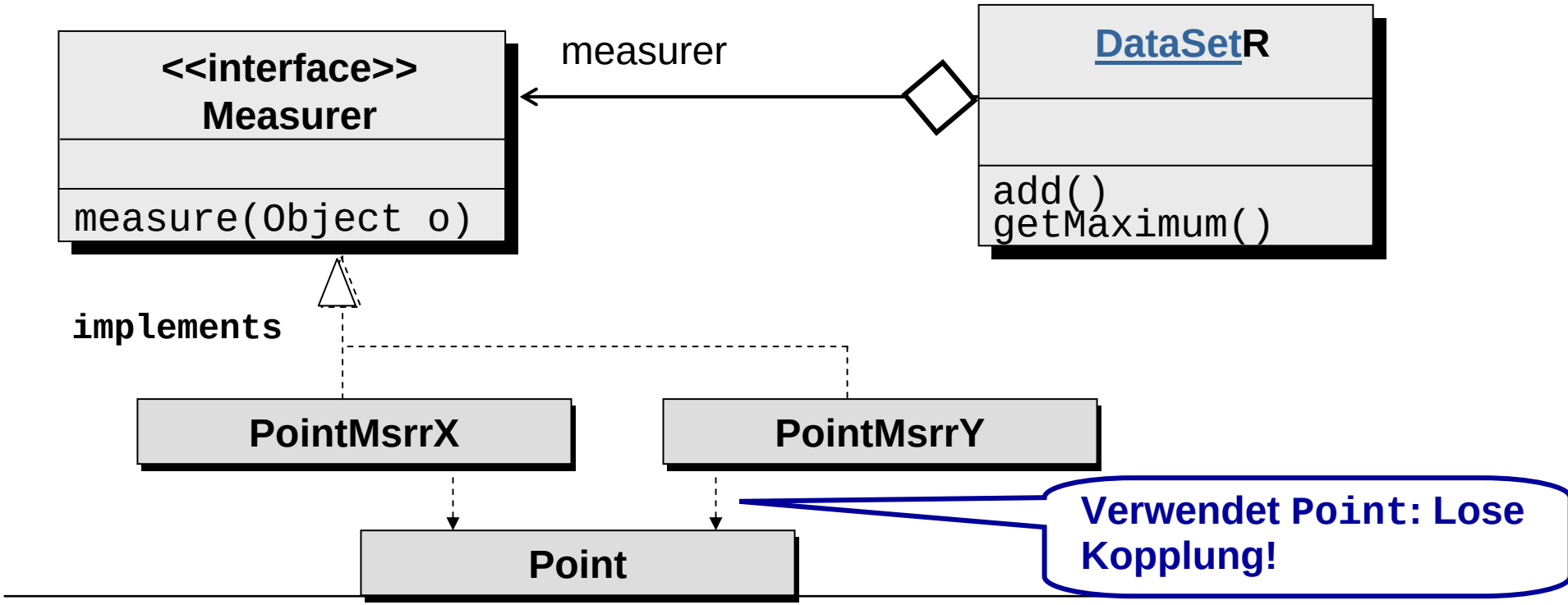
Ergänzung von BankAccount um getMeasure

```
public class Point implements Measurable
{       ... <alte Point-Implementierung>
        public double getMeasure()
        {       return Math.sqrt(getX()*getX() + getY()*getY());
        }
}
```

Ergänzung von Point um getMeasure

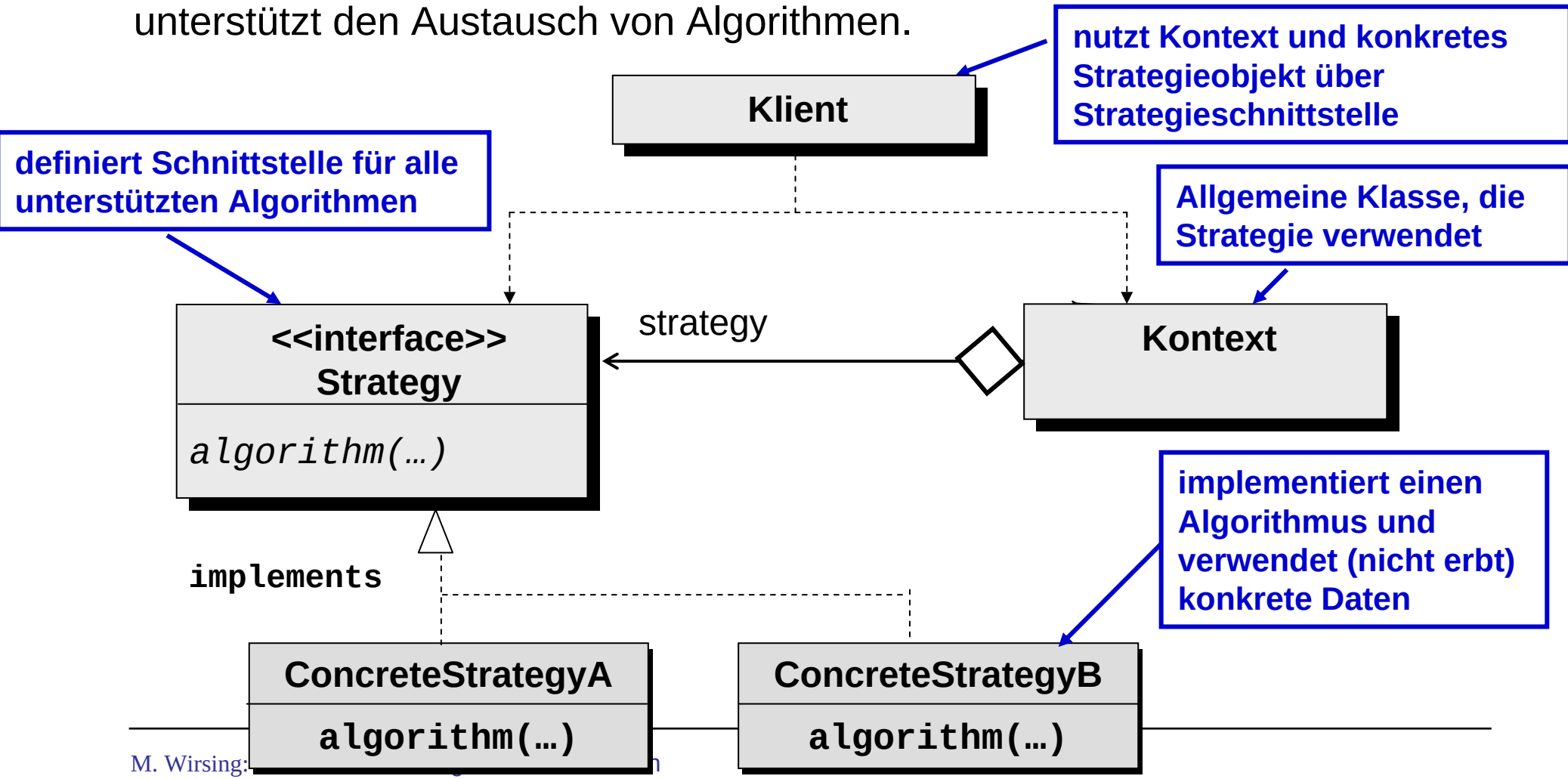
(c) Schnittstelle für mehrere Messoperationen

Das Interface `Measurer` unterstützt die Verwendung von mehreren Messoperationen auf der gemessenen Klasse `Point`; die Messklassen implementieren die Operation `measure(Object o)` und benützen `Point`. Allgemein spricht man von **Strategie-Muster.**



Das Strategiemuster

- Das **Strategiemuster** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.



(c) Beispiel Strategiemuster: Schnittstelle für mehrere Messoperationen

```
public interface Measurer
{  double measure(Object anObject) ;
}

public class DataSetR
{       private double sum;
        private Object maximum;
        private int count;
        private Measurer measurer ;
        public DataSet(Measurer aMeasurer)
        {       sum = 0; count = 0; maximum = null;
                measurer = aMeasurer;}
        public void add(Object x)
        {       sum = sum + measurer.measure(x) ;
                if (count == 0
                   || measurer.measure(maximum) < measurer.measure(x))
                maximum = x;
                count++;   } ...
```

Zu messendes Objekt

Allgemeine Klasse (Kontext), die `Measurer` verwendet

Beispiel Strategiemuster: Implementierungen der Schnittstelle Measurer

Benützt `Point`

```
public class PointMsrrX implements Measurer
{       public double measure(Object anObject)
        {       Point aPoint = (Point)anObject;
                return aPoint.getX();
        }
}
```

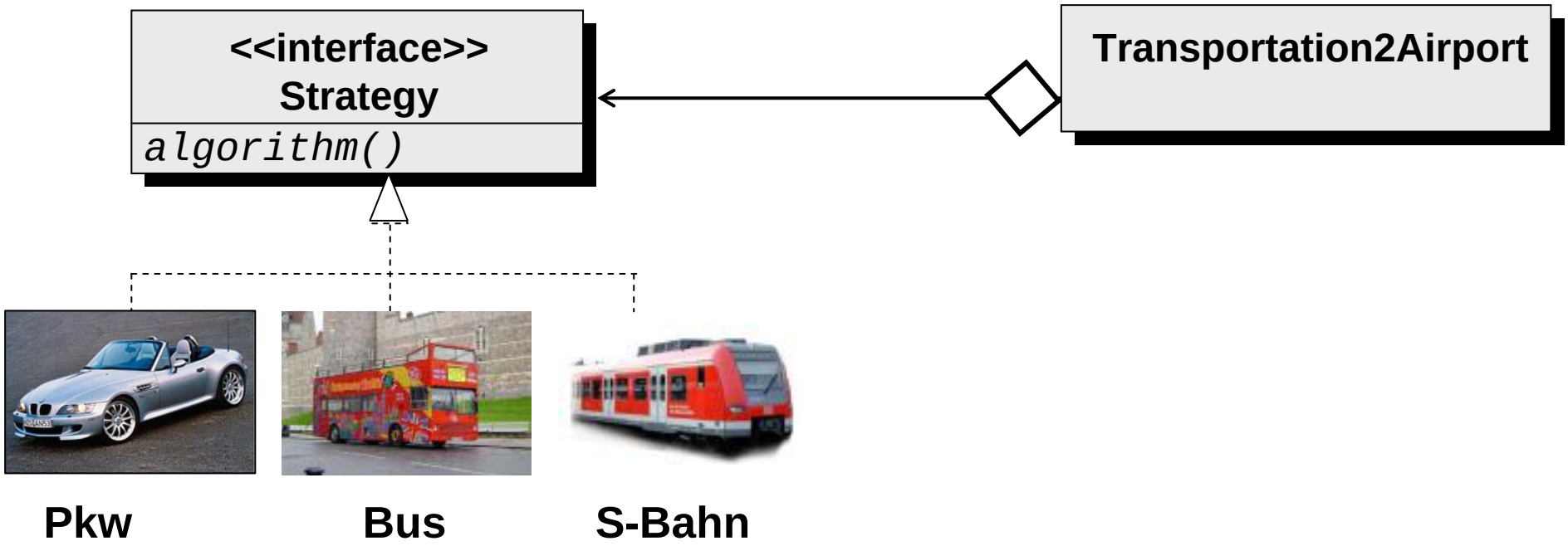
Typanpassung: nur Point-Instanzen sind korrekte aktuelle Parameter

```
public class PointMsrrY implements Measurer
{       public double measure(Object anObject)
        {       Point aPoint = (Point)anObject;
                return aPoint.getY();
        }
}
```

Beispiel: Verwendung von PointMsrrX

Das Strategiemuster: Weitere Beispiele

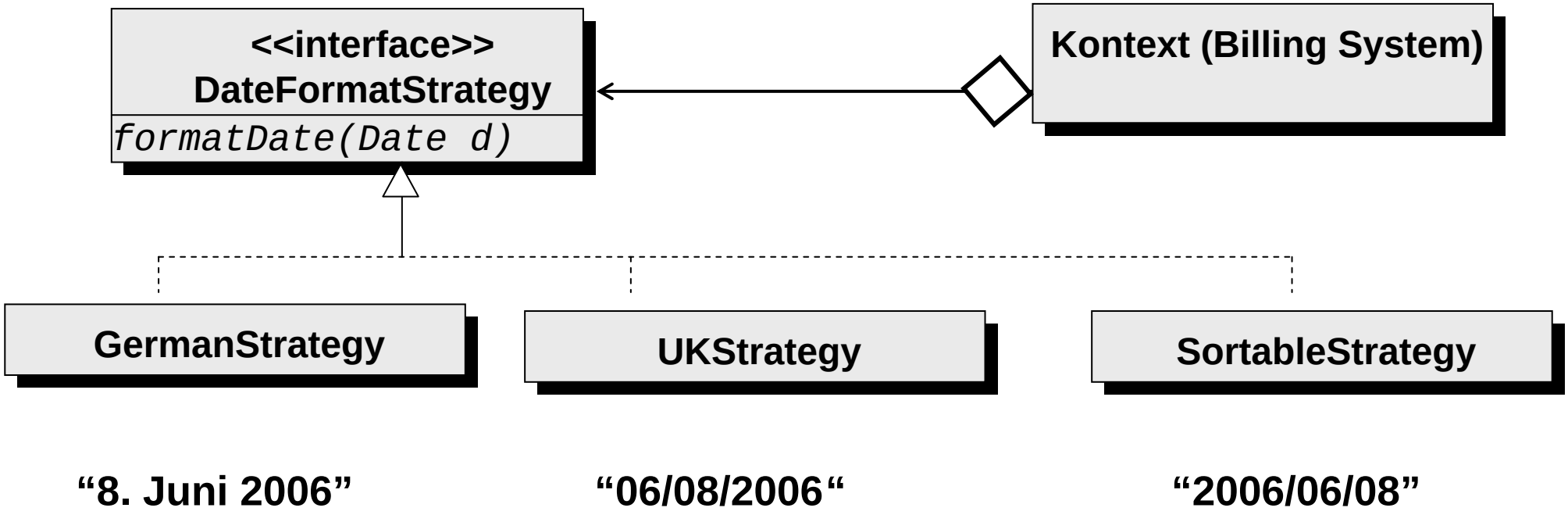
- **Transport zum Flughafen:**
  Strategien (Optionen) können sein: S-Bahn, Bus, Pkw



Das Strategiemuster: Weitere Beispiele

- **Datumsformate:**

  In Deutschland und England werden für das gleiche Datum unterschiedliche Formate verwendet.

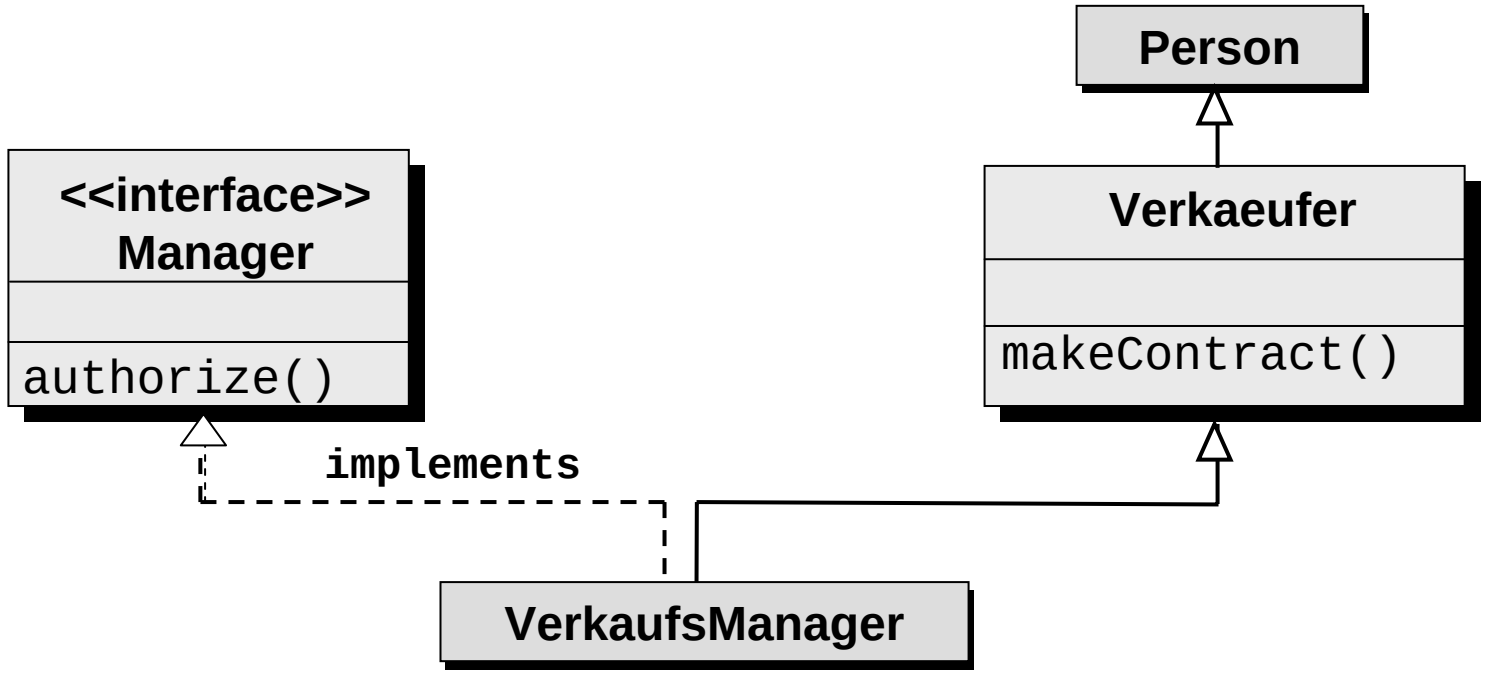


Das Strategiemuster: Vorteile/Nachteile

- Das **Strategiemuster** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.
- **Vorteile**
  - Es wird eine Familie von Algorithmen definiert.
  - Strategien bieten eine Alternative zur Unterklassenbildung,
    helfen Mehrfachverzweigungen zu vermeiden und
    verbessern dadurch die Wiederverwendung.
  - Strategien ermöglichen die Auswahl aus verschiedenen Algorithmen-Implementationen ("Algorithmen-Polymorphie") und erhöhen dadurch die Flexibilität.
- **Nachteile**
  - Klienten müssen die unterschiedlichen Strategien kennen,
    um zwischen ihnen auswählen zu können.
  - Gegenüber der direkten Implementation der Algorithmen im Klienten erzeugen
    Strategien zusätzlichen Kommunikationsaufwand zwischen Strategie und Klient.
  - Die Anzahl der Objekte wird erhöht.

Schnittstellen zur Simulation mehrfacher Vererbung

Die mehrfache Vererbung des Verkaufsmanagers läßt sich in Java durch eine Schnittstelle simulieren:



Hier muss `authorize()` in `VerkaufsManager` implementiert werden

Bemerkung: Verwendung von Schnittstellen

- Konstanten in Schnittstellen sind automatisch **„public final“.**

Beispiel:

```
public interface ComparableConstants
{    boolean IST_GROESSER = true;
     boolean IST_KLEINER = false;
}
```

- Die in Schnittstellen deklarierten Methoden sind **public**!

```
public interface I
{       void m();
 ...
}
```

Jede Implementierung von `I` muss `m` als **public** deklarieren:

```
public class C implements I
{       public void m();
 ...
}
```

Zusammenfassung

- **Mehrfache Vererbung** ist in Java nur für Schnittstellen, aber nicht für Klassen erlaubt. Mehrfache Vererbung bei Klassen kann aber mit Hilfe von Schnittstellen („`interface`“) simuliert werden.
- **Pakete** dienen zur Strukturierung großer Programmsysteme; sie fassen verwandte Klassen zusammen.
- Eine **Schnittstelle** deklariert abstrakte Methoden (Methodenköpfe) und Konstanten. Eine Klasse **implementiert eine Schnittstelle**, wenn sie alle Methoden der Schnittstelle implementiert. Schnittstellen können mehrfach von anderen Schnittstellen erben.
- Schnittstellen unterstützen die Wiederverwendung allgemeiner Klassen durch Entkopplung der „Parameterklassen“.
- Das **Strategiemuster** entkoppelt Objekte von ihrem Verhalten und unterstützt den Austausch von Algorithmen.