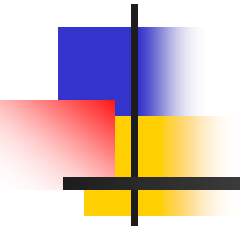


6. (Entwurfs-)Muster

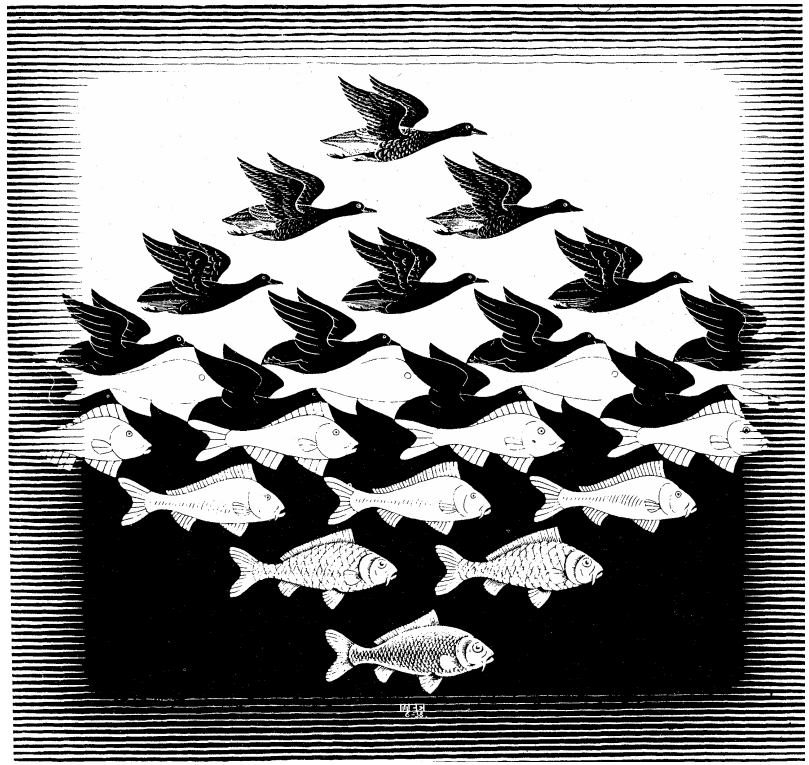
von
Prof. Dr. Wolfgang Weber

[Wi05, S. 355f], [GHJV96]

Muster (Patterns)

„Auf der Grundlage von Erfahrung und der Reflexion darüber können wir wiederkehrende Muster erkennen ...

Einmal erkannte Muster leiten unsere Wahrnehmung.“



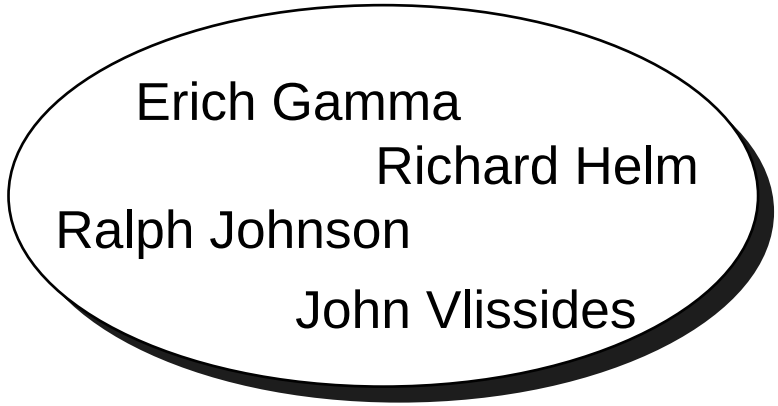
Arten von Mustern

- Architekturmuster (Architectural Patterns)
  - dokumentieren Entwurfserfahrungen, die sich auf globale Aspekte des Gesamtsystems beziehen
  - Sie kennen: 4-Schichten-SW-Architektur
- Entwurfsmuster (Design Patterns)
  - machen lokale, sich auf wenige Klassen beziehende Entwurfserfahrungen der Wiederverwendung zugänglich
  - nur abstrakt (UML-Modelle)
  - keine Implementierung (höchstens Code-Beispiele)
- Rahmenwerke (Frameworks)
  - erlauben Wiederverwendung von ganzen Entwürfen
  - mit teilweiser Implementierung
  - umfassen wesentliche Aspekte einer Gruppe ähnlicher Anwendungssysteme
  - Sie kennen: Visual C++

Entwurfsmuster

„Design Patterns – Elements of Reusable Object-Oriented Software“ (1994)
deutsch: [GHJV96]

„Gang-of-Four“

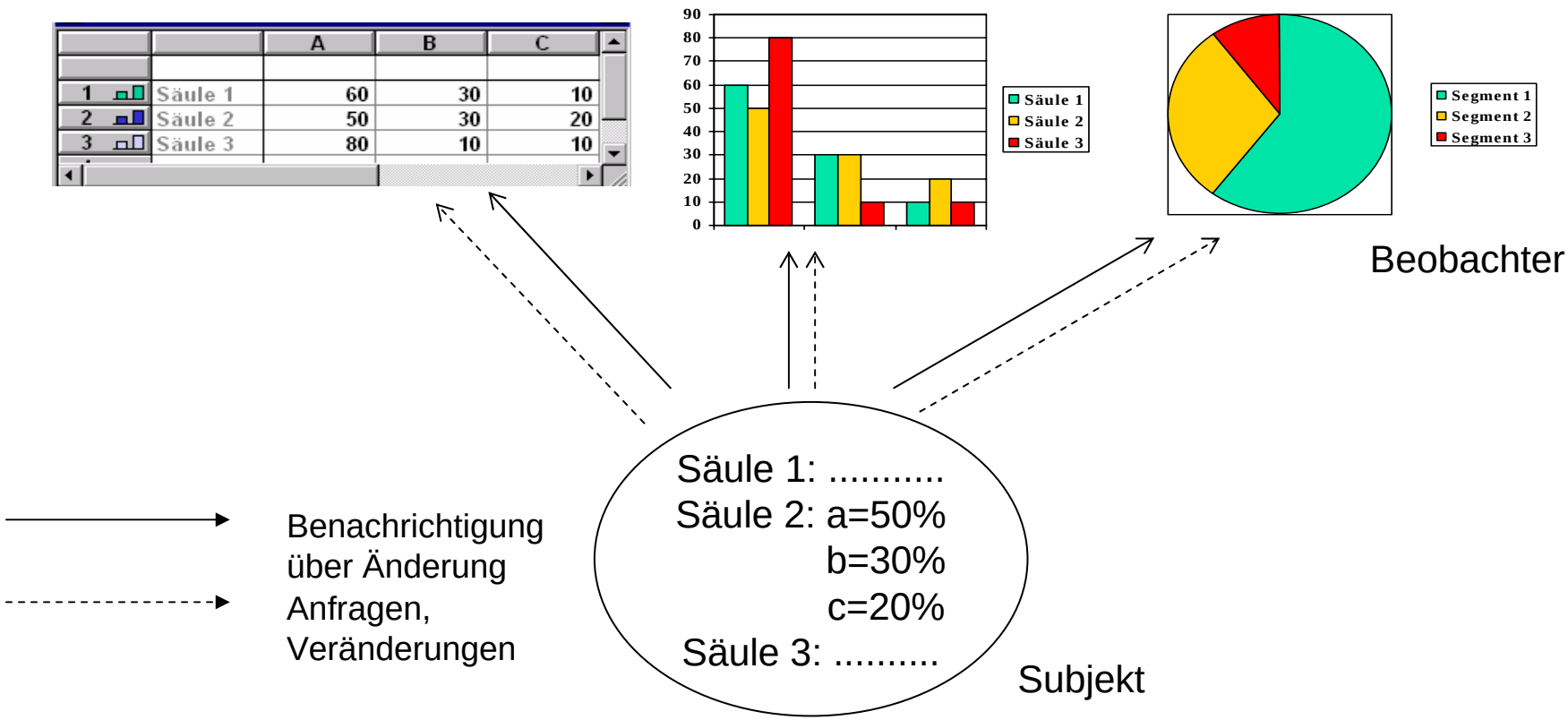


> helfen, existierende SW-Entwürfe zu analysieren und zu reorganisieren,

> sind „Mikroarchitekturen“, die sich innerhalb größerer SW-Architekturen strukturerhaltend auf die Implementierung abbilden lassen

Beobachtermuster (Observer Pattern)

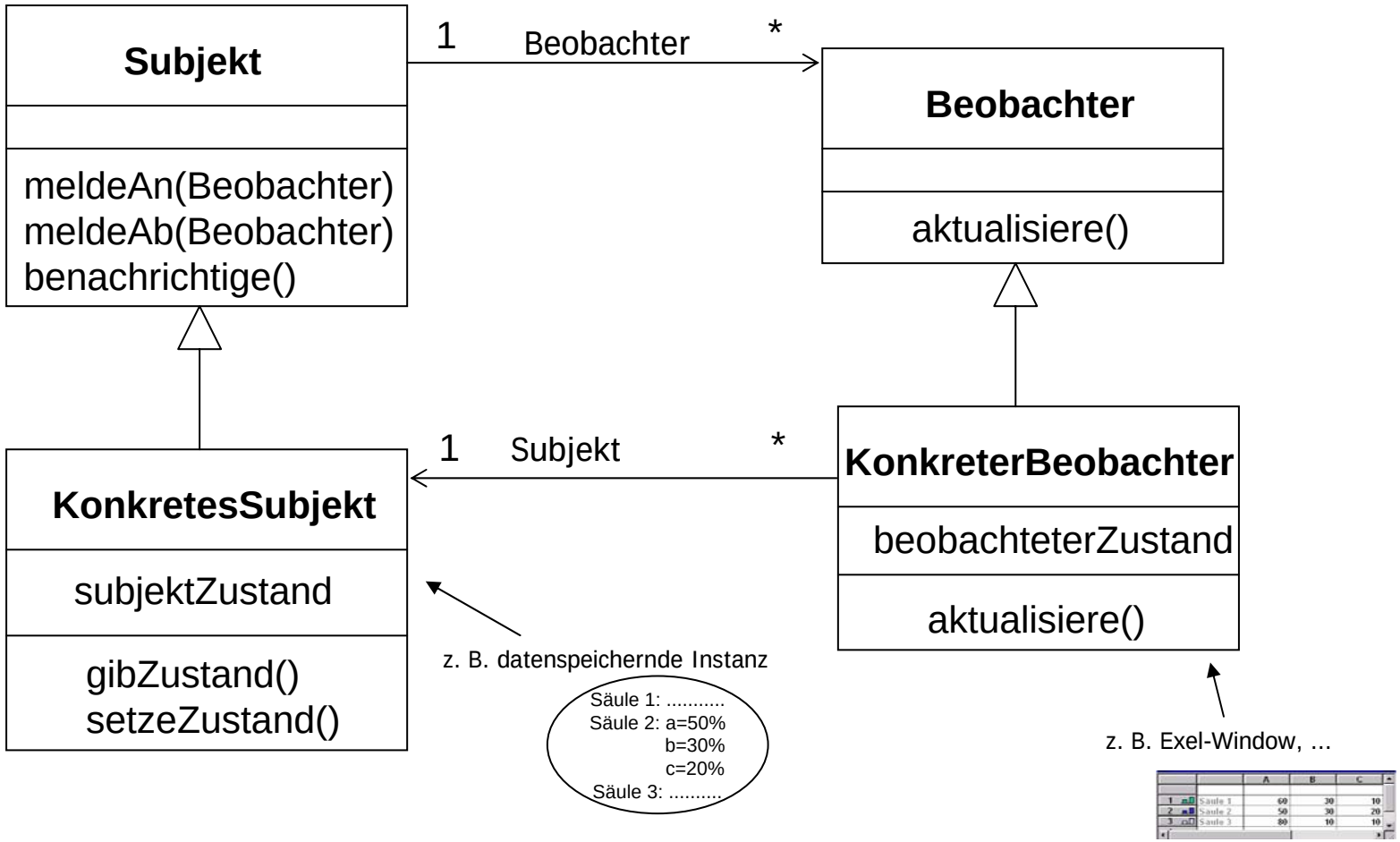
Motivation:



Die drei Beobachter-Objekte verhalten sich so, als würden sie einander kennen: sie hängen von einem gemeinsamen Datenobjekt ab. $\Rightarrow$ Es ist notwendig, die drei Objekte über Zustandsänderungen des Datenobjektes zu informieren.

Beobachtermuster (Observer Pattern)

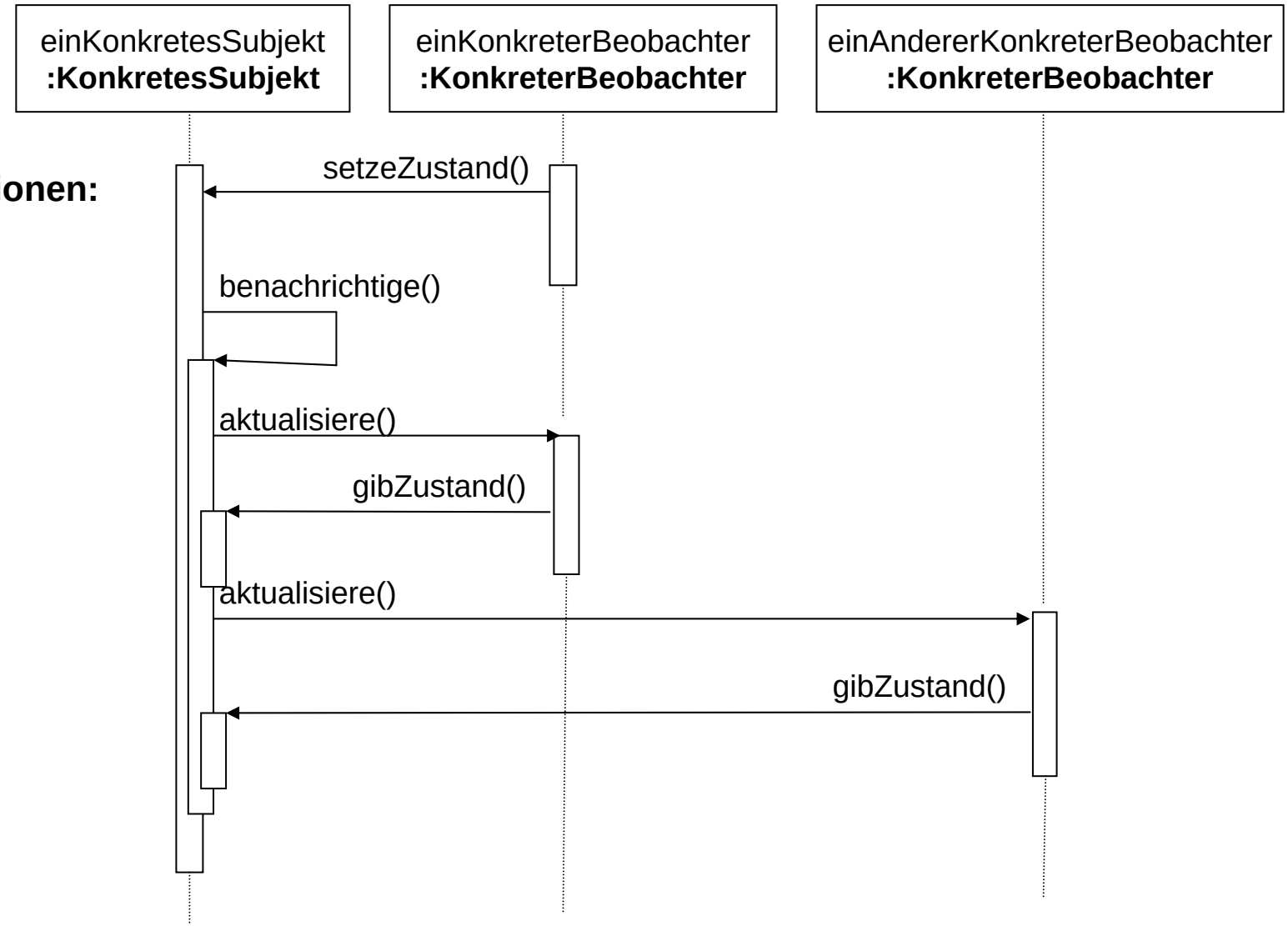
Struktur:



Bei Veränderung => benachrichtige(): für alle b in beobachter{b->aktualisiere}
Nach Benachrichtigung => aktualisiere(): {beobachteter Zustand = subjekt->gibZustand}

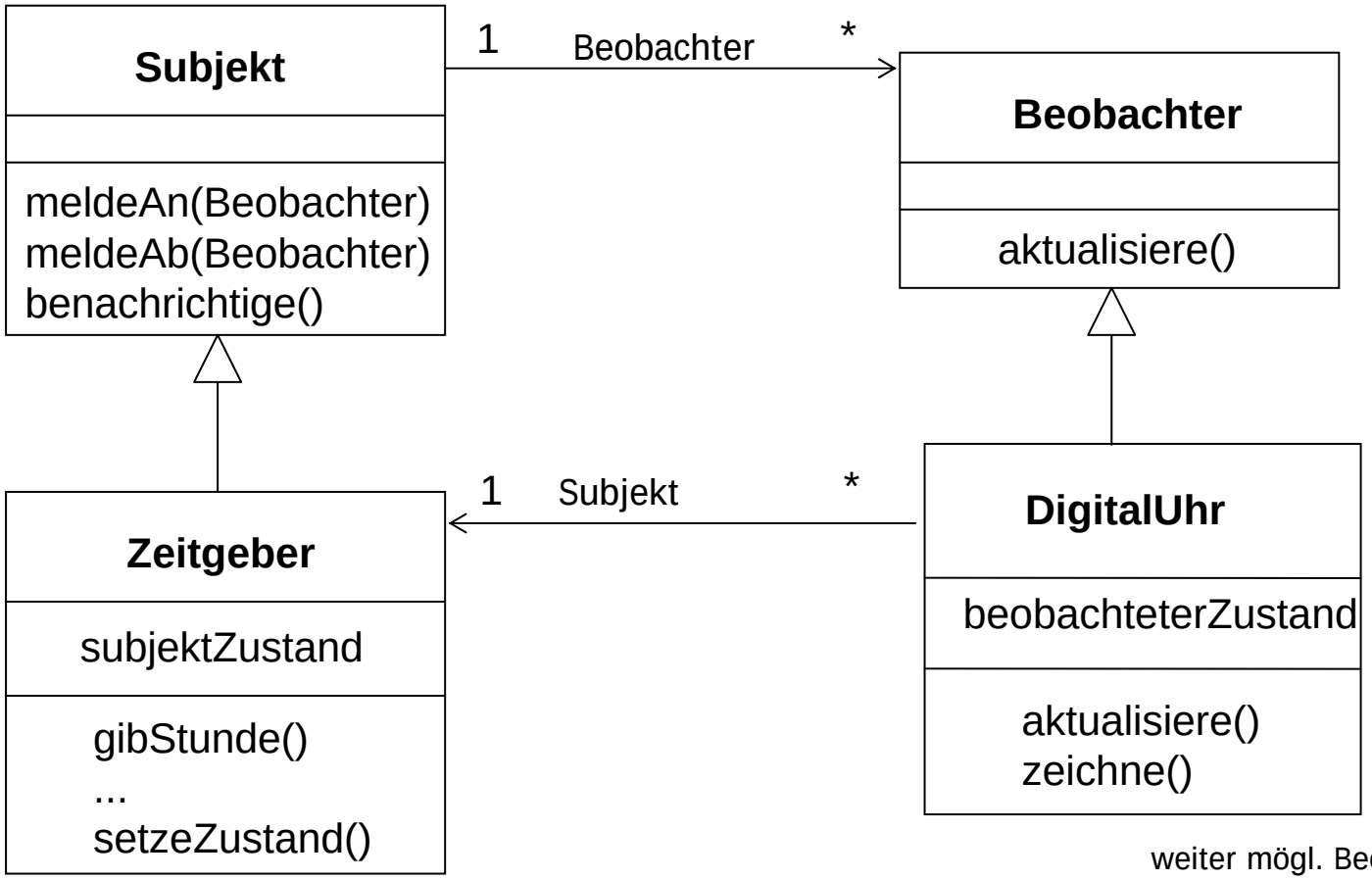
Beobachtermuster (Observer Pattern)

Interaktionen:



Beobachtermuster – Beispiel Digitaluhr

Struktur:



weiter mögl. Beobachter:
z. B. Analoguhr

Beobachtermuster – Beispielcode Digitaluhr

```
class Subjekt{
public:
 virtual ~Subjekt();
 virtual void meldeAn(Beobachter*);
 virtual void meldeAb(Beobachter*);
 virtual void benachrichtige();
protected:
 Subjekt();
private:
 Liste<Beobachter*>* _beobachter;
};

void Subjekt::meldeAn(Beobachter* beobachter) {
 _beobachter->haengeAn(beobachter);
}

void Subjekt::meldeAb(Beobachter* beobachter) {
 _beobachter->entferne(beobachter);
}

void Subjekt::benachrichtige() {
 ListenIterator<Beobachter*> iter(_beobachter);
 for(iter.start(); !iter.istFertig(); iter.weiter())
 {
    iter.aktuellesElement()->aktualisiere(this);
 }
}
```

```
class ZeitGeber : public Subjekt {
public:
 ZeitGeber();
 virtual int gibStunde();
 virtual int gibMinute();
 virtual int gibSekunde();
 void tick();
};
```

```
class Beobachter {
public:
 virtual ~Beobachter();
 virtual void aktualisiere(Subjekt*
           dasGeaenderteSubjekt)=0;
protected:
 Beobachter();
};
```

```
class DigitalUhr : public Widget,
                   public Beobachter{
public:
 DigitalUhr(ZeitGeber* subjekt){
   _subjekt=subjekt; _subjekt->meldeAn(this)};
 virtual ~DigitalUhr(){...meldeAb(this)};
 virtual void aktualisiere(Subjekt* _subjekt){
   zeichne()};
virtual zeichne(){
   int stunde=_subjekt->gibStunde; ... }
private:
 ZeitGeber* _subjekt;
};
```

Entwurfsmuster

Der Patternkatalog von [GHLV96] gliedert sich in

> Erzeugungsmuster
z. B. „Abstrakte Fabrik“ – Objektgeneratoren

> Strukturmuster
z. B. „Kompositum“ – Zusammenfügen einzelner Objekte zu Baumstrukturen

> Verhaltensmuster
z. B. „Beobachter“ – Benachrichtigung aller assoziierten Objekte in einer 1:n-Beziehung, sobald sich ein Objekt-Zustand verändert
z. B "Das Strategie-Muster" – Kapselung von Algorithmen

Wie beschreibt man Entwurfsmuster?

- **Musternamе**
  - vermittelt knapp und präzise den wesentlichen Gehalt des Musters.
  - Ein guter Name ist wichtig, weil er Teil des Entwurfsvokabulars werden sollte.
- **Zweck**
  - Was macht das Entwurfsmuster?
  - Was ist sein Grundprinzip und was sein Zweck?
  - Welche spezifischen Fragestellungen oder Probleme werden behandelt?
- **Auch bekannt als**
  - Andere wohlbekannte Namen für das Muster, sofern es sie gibt.
- **Motivation**
  - Beispiel, das ein Entwurfsproblem schildert und zeigt, wie die Klassen- und Objektstrukturen des Musters das Problem lösen.
- **Anwendbarkeit**
  - in welchen Situationen kann das Entwurfsmuster angewandt werden.
  - Benennt die Problemsituationen, in denen das Muster helfen kann.
  - Woran kann man diese Situationen erkennen.
- **Struktur**
  - Grafischen Repräsentation der Klassen im Muster.
  - Darstellung mit Klassendiagrammen der UML.

Wie beschreibt man Entwurfsmuster?

- **Teilnehmer**
  - Beschreibung der am Entwurfsmuster beteiligten Klassen und Objekte sowie ihre Zuständigkeiten.
- **Interaktionen**
  - Beschreibung, wie die Teilnehmer zur Erfüllung der gemeinsamen Aufgabe zusammenarbeiten.
- **Konsequenzen**
  - Vor- und Nachteile sich durch die Anwendung des Musters ergeben.
  - Welche Ergebnisse sind zu erwarten
- **Implementierung und Beispielcode**
  - Beispielhafter Quellcode / Codefragmente
  - Hinweise auf Fallen.
  - Tipps, anwendbare Techniken
- **Bekannte Verwendungen**
  - Beispiele als Muster auf, die in echten Systemen zu finden sind.
- **Verwandte Muster**
  - Muster wird in Bezug zu anderen Mustern gesetzt.
  - Diskutiert die Unterschiede.
  - Erläutert, welche Muster zusammen verwendet werden können.

Das Kompositum-Muster (Composite Pattern)

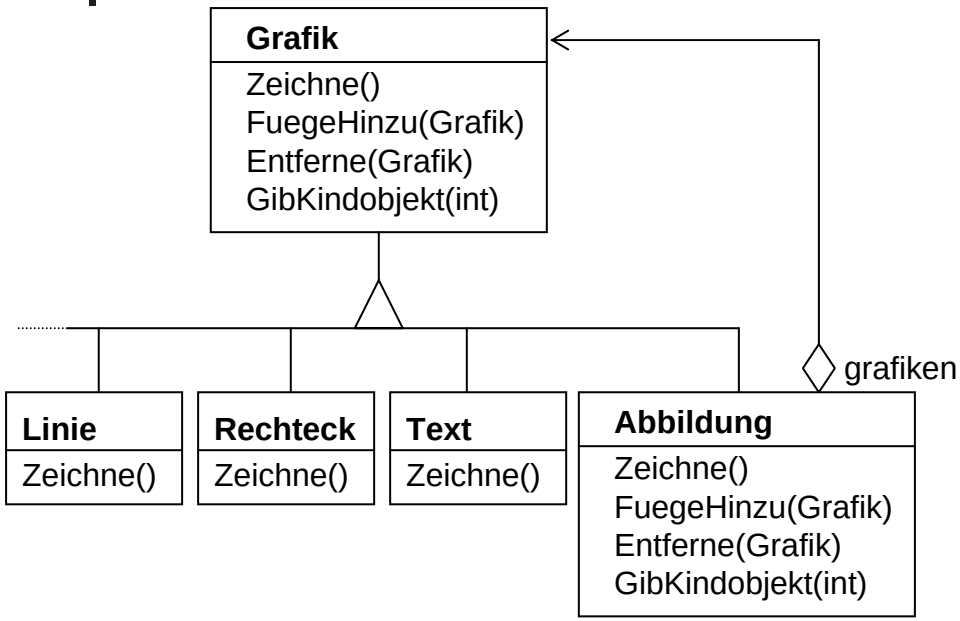
Zweck:

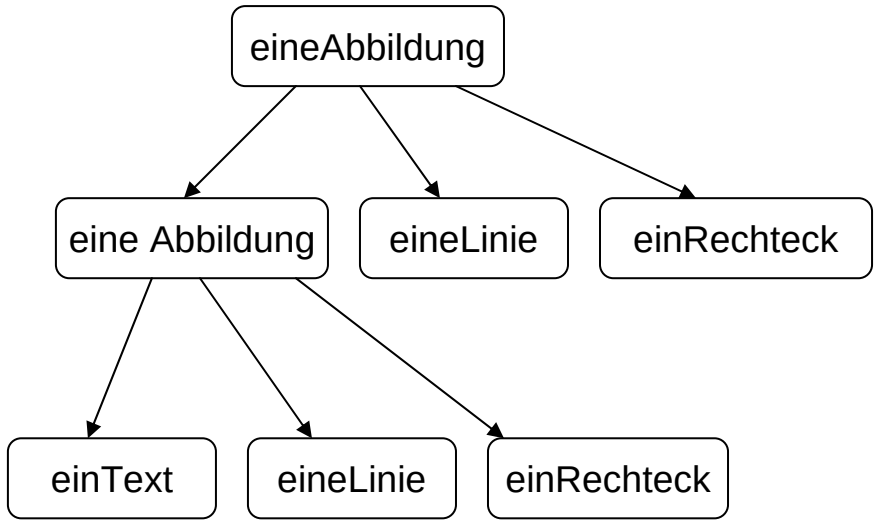
- Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren.
- Ermöglicht es Klienten, sowohl einzelne Objekte (Blätter eines Baumes) als auch Kompositionen von Objekten (Nicht-Blatt-Knoten) einheitlich zu behandeln.

Motivation:

- Aufbau komplexer Diagramme aus einfachen Komponenten, z.B. grafische Anwendungen.
- Der Benutzer kann aus „elementaren“ Komponenten (Blätter eines Baumes) größere Komponenten zusammenfügen und aus diesen wiederum größere Komponenten etc.
- ***Idee***: Ein Client sollte bei der Verwendung der Komponenten nicht differenzieren müssen, ob es sich um eine elementare oder eine zusammengesetzte Komponente handelt.

Das Kompositum-Muster





eine rekursive Struktur aus Grafikobjekten

Zeichne(): für alle g in Kindobjekten: g.Zeichne

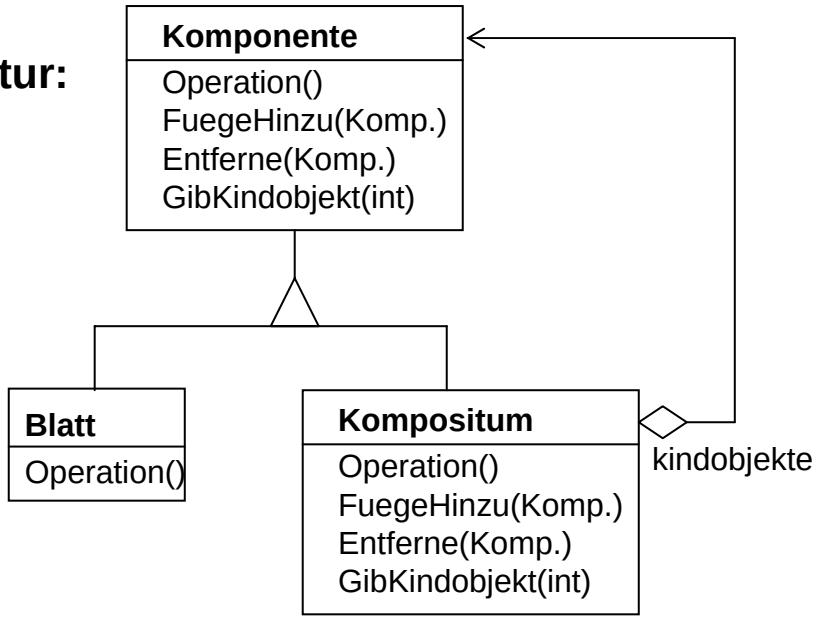
FügeHinzu(Graphik g): Füge g in Liste der Grafikobjeke ein.

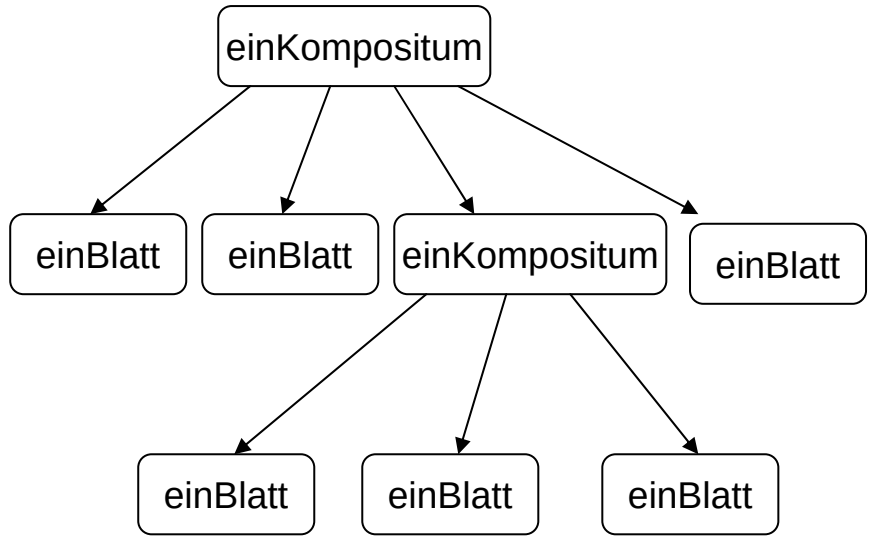
Anwendbarkeit

- Repräsentation von Teil-Ganzes-Hierarchien
- Klient soll in der Lage sein, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren
- Klient soll alle Objekte im Kompositum einheitlich behandeln können.

Das Kompositum-Muster

Struktur:





rekursive Struktur aus Komponenten

Teilnehmer:

- **Komponente:** Schnittstellendeklarationen / -Implementierungen von Defaultverhalten.
- **Blatt:** Besitzt keine Kindobjekte,
  definiert Verhalten für die primitiven Objekte in der Komposition.
- **Kompositum:** Definiert Verhalten für Objekte, die Kindobjekte haben können,
  speichert Kindobjektkomponenten,
  implementiert entsprechende Operationen der Schnittstelle von Komponenten.
- **Klient:** Manipuliert die Objekte in der Komposition durch die Schnittstelle von Komponenten.

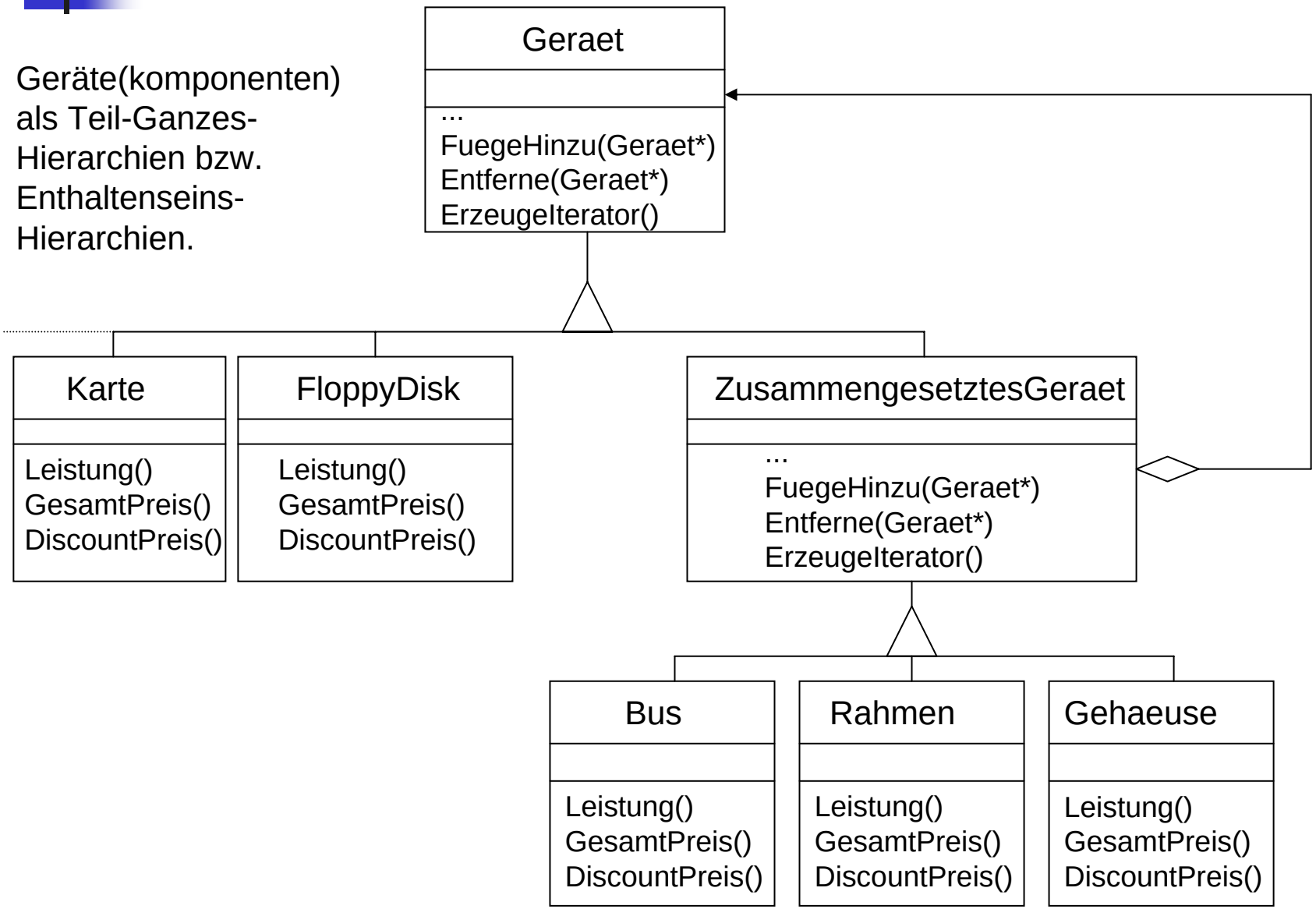
Das Kompositum-Muster

Interaktionen:

- Klienten verwenden Schnittstellen der Klassenkomponente, um mit Objekten der Komponentenstruktur zu interagieren.
- Wenn Empfänger = Blatt
  - ⇒ Direkte Abhandlung der Anfrage.
- Wenn Empfänger = Kompositum
  - ⇒ Weiterleitung der Anfrage an Kindobjekte

Das Kompositum-Muster - Beispiel-Klassendiagramm

Geräte(komponenten) als Teil-Ganzes-Hierarchien bzw. Enthaltenseins-Hierarchien.



Das Kompositum-Muster - Beispielcode (1)

```
class Geraet {
public:
   virtual ~Geraet();
   const char* Name()
         {return _name;}
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis();
   virtual FuegeHinzu(Geraet*);
   virtual Entferne(Geraet*);
   virtual Iterator<Geraet*>*
           ErzeugeIterator();
protected:
   Geraet(const char*);
private:
   const char* _name;  };
```

```
class ZusammengesetztesGeraet:
      public Geraet {
public:
   virtual ~ZusammengesetztesGeraet();
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis();
   virtual FuegeHinzu(Geraet*);
   virtual Entferne(Geraet*);
   virtual Iterator<Geraet*>*
           ErzeugeIterator();
protected:
   ZusammengesetztesGeraet(const char*);
private:
   Liste<Geraet*> _teile;  };
```

S. 18

```
class FloppyDisk:public Geraet {
public:
   FloppyDisk(const char*);
   virtual ~FloppyDisk();
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis(); };
```

```
class Gehaeuse:
   public ZusammengesetztesGeraet {
public:
   Gehaeuse(const char*);
   virtual ~Gehaeuse();
   virtual Watt Leistung();
   virtual Betrag GesamtPreis();
   virtual Betrag DiscountPreis(); };
```

Das Kompositum-Muster - Beispielcode (2)

```
class ZusammengesetztesGeraet:
      public Geraet {

public:
  ...
  virtual Betrag GesamtPreis();
  ...
};
```

```
Betrag ZusammengesetztesGeraet::GesamtPreis() {
  Iterator<Geraet*>* iter = ErzeugeIterator();
  Betrag gesamt = 0;

  for (iter->Start(); !iter->IstFertig(); iter->Weiter())
     gesamt += iter->AktuellesElement() -> GesamtPreis();
  delete iter;
  return gesamt;
}
```

```
...
Gehaeuse* gehaeuse = new Gehaeuse("PC Gehaeuse");

Rahmen*   rahmen   = new Rahmen("PC Rahmen");
gehaeuse->FuegeHinzu(rahmen);

Bus* bus = new Bus("MCA Bus");
bus->FuegeHinzu(new Karte("16Mbs Token Ring"));

rahmen->FuegeHinzu(bus);
rahmen->FuegeHinzu(new FloppyDisk("3.5in Floppy"));

cout << "Der Gesamtpreis betraegt“ << gehaeuse->GesamtPreis()
   << endl;
...
```

Das Strategie-Muster (Strategy Pattern)

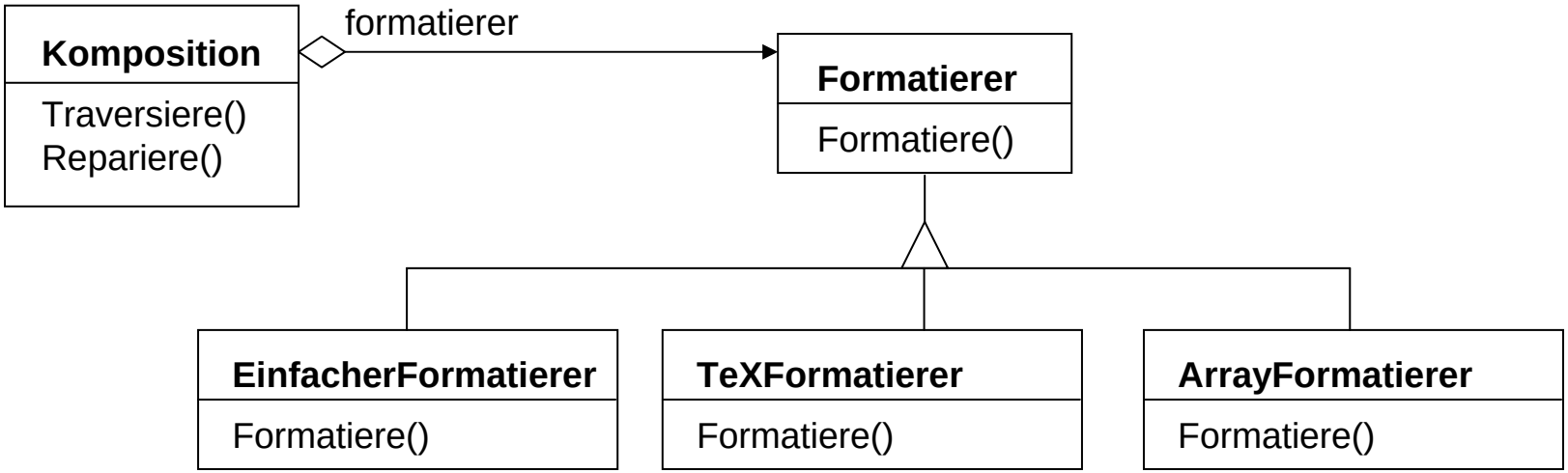
Zweck:

- Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar.
- Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von den ihn nutzenden Klienten zu variieren.

Motivation:

- Es gibt unterschiedlichste Algorithmen zum *Zeilenumbruch in einem Textstrom*.
- Es ist nicht wünschenswert, alle möglichen Algorithmen in einer Klasse fest zu codieren.
- ***Idee*:** Jeder Algorithmus wird (als Strategie) in einer Klasse gekapselt und ist über eine (abstrakte) Superklasse, die eine geeignete Schnittstelle zur Verfügung stellt, anwendbar.

Das Strategie-Muster



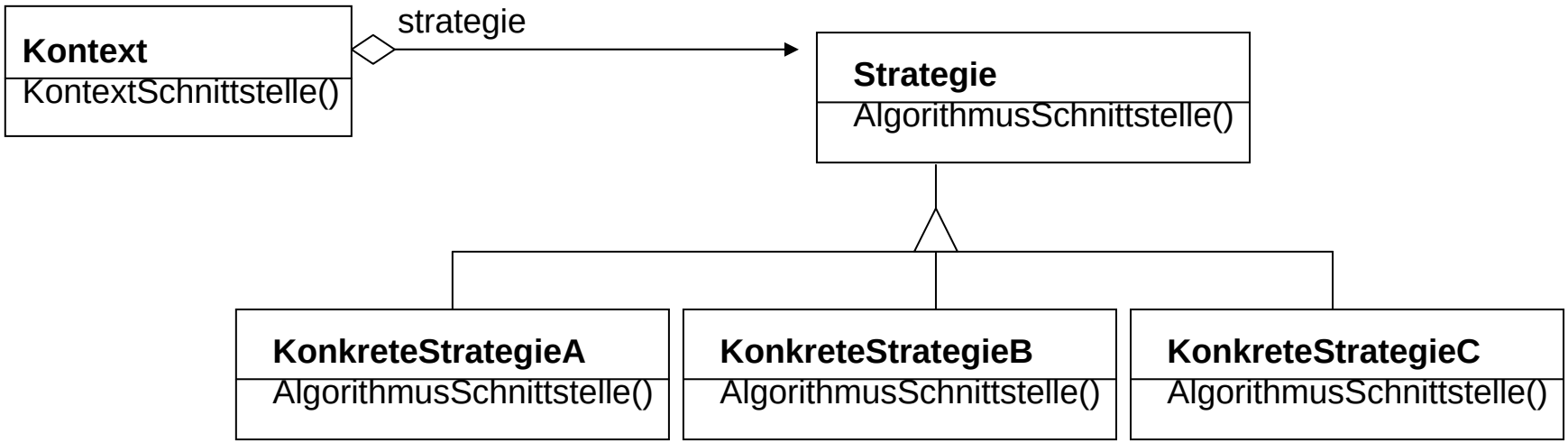
Repariere(): formatierer->Formatiere()

Anwendbarkeit

- Viele verwandte Klassen unterscheiden sich nur in ihrem Verhalten.
- Unterschiedliche Varianten eines Algorithmus werden benötigt, z.B. mit unterschiedlichen Vor- und Nachteilen bzgl. der Geschwindigkeit und des Speicherplatzverbrauchs.
- Daten, die für den Algorithmus relevant sind, aber dem Klienten nicht bekannt sein sollen, können in einer Strategie „verborgen“ werden.

Das Strategie-Muster

Struktur:



Teilnehmer:

- **Strategie:** Schnittstellendeklarationen für alle unterstützten Algorithmen
- **Konkrete Strategie:** Implementiert den Algorithmus unter Verwendung der Strategieschnittstelle
- **Kontext:** Wird mit einem KonkreteStrategieX-Objekt konfiguriert, verwaltet eine Referenz auf ein Strategieobjekt und kann eine Schnittstelle für den Zugriff auf seine Daten definieren.

Das Strategie-Muster

Konsequenzen: Strategien ersetzen Bedingungsanweisungen.
Bei Hinzufügung einer konkreten Strategie-> nur kleine Änderung in Komposition:

```
void Komposition::Repariere() {
  switch(_umbruchStrategie) {
  case EinfacheStrategie:
    FormatiereMitEinfacherStrategie();
    break;
  case TeXStrategie:
    FormatiereMitTeXStrategie();
    break;
  // ...
  }
  // Führe die Ergebnisse mit der exi-
  // stierenden Komposition zusammen
}
```

Beispielcode für Zeilenumbruch ohne Strategieobjekt

Die Möglichkeiten für einen Zeilenumbruch werden an das Strategieobjekt delegiert.

```
void Komposition::Repariere() {
  _formatierer->Formatiere();
  // Führe die Ergebnisse mit der exi-
  // stierenden Komposition zusammen
}
```

Das Strategie-Muster - Beispielcode (1)

```
class Komposition {

public:
   Komposition(Formatierer*);
   void Repariere();
```
S. 24
```
private:
   Formatierer* _formatierer;
   //Liste von Komponenten (Texte,
   //grafische Elem. ...):
   Komponente* _komponenten;
   //Anzahl der Komp. in der Liste:
   int _komponentenAnzahl;
   //Zeilenbreite in der Komposition:
   int _zeilenBreite;
   //Position der Zeilenumbrüche in
   //den Komponenten:
   int* _zeilenUmbrueche;
   //Anzahl der Zeilen
   int _zeilenAnzahl;
};
```

```
class Formatierer {
public:
   virtual int Formatiere(Koordinate ausmasse[],
      Koordinate dehnbarkeit[],
      Koordinate stauchbarkeit[],
      int komponentenAnzahl, int ZeilenBreite,
      int umbrueche[]) = 0;
protected:
   Formatierer();
};
```

...

```
class ArrayFormatierer : public Formatierer {
public:
   ArrayFormatirer(int breite);

   virtual int Formatiere(Koordinate ausmasse[],
      Koordinate dehnbarkeit[],
      Koordinate stauchbarkeit[],
      int komponentenAnzahl, int ZeilenBreite,
      int umbrueche[]) = 0;
   // ...
};
```

...

Das Strategie-Muster - Beispielcode (1)

```
void Komposition::Repariere() {
   Koordinate* ausmasse;
   Koordinate* dehnbarkeit;
   Koordinate* stauchbarkeit;
   int* umbrueche;

   // Vorbereitung der Arrays mit den
   // gewuenschten Groessen fuer die
   // Komponenten
   // ...

   // Position der Umbrueche wird
   // bestimmt
   int umbruchAnzahl;
   umbruchAnzahl=
   _formatierer->Foramtiere(ausmasse,
   dehnbarkeit, stauchbarkeit,
   komponentenAnzahl,_zeilenBreite,
   umbrueche);

   // Darstellung der Komponente
   // gemaess den Umbruechen
   // ...
}
```

⇒ Instanziierung verschiedener Kompositions-Objekte:

```
Komposition* schnell =
   new Komposition(new EinfacherFormatierer);

Komposition* schick =
   new Komposition(new TeXFormatierer);

Komposition* icons =
   new Komposition(new ArrayFormatierer);
```

vgl. S. 23

The Blob  - ein Software Development AntiPattern

Generelle Form

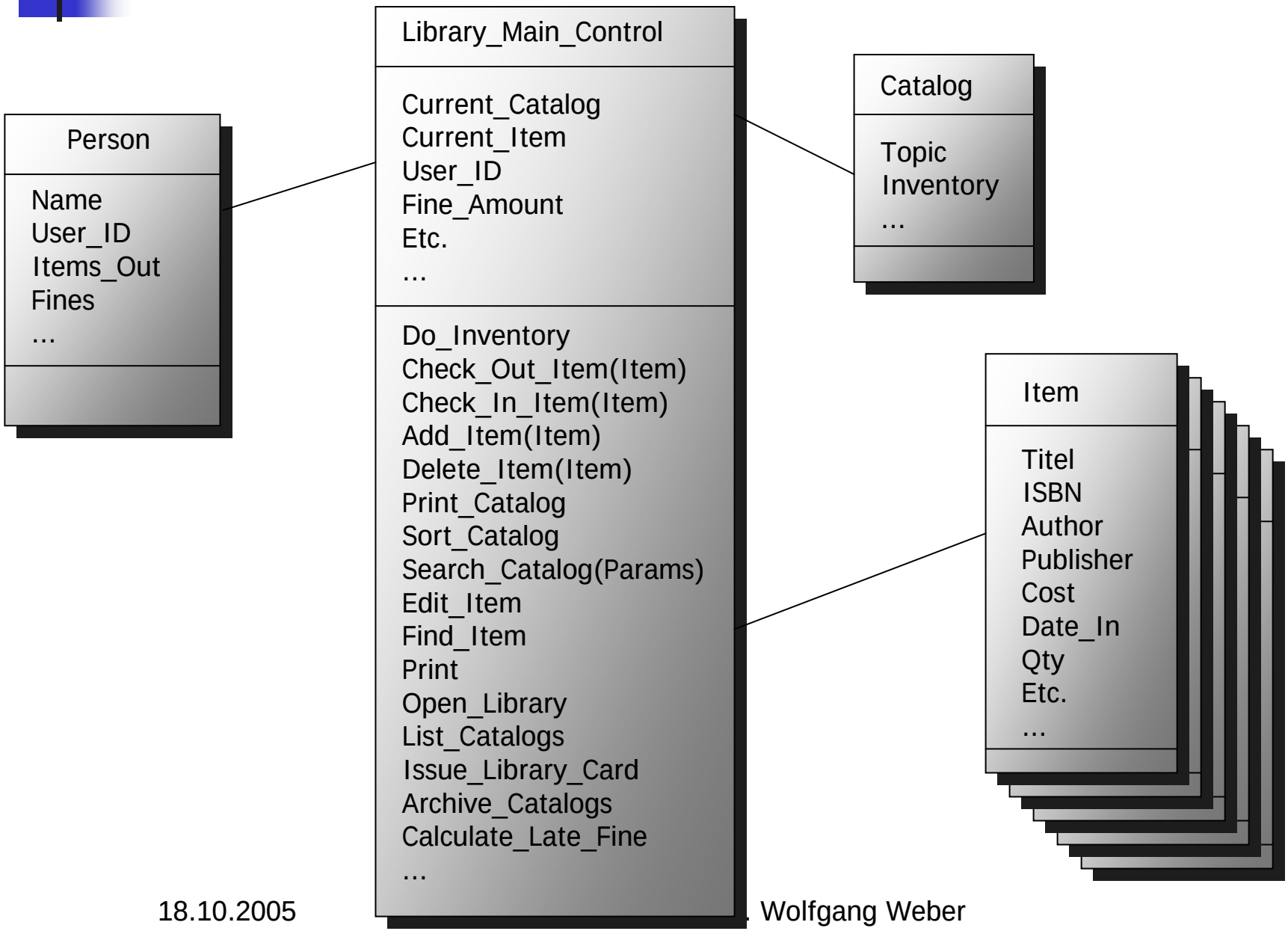
- Typisch für den Blob ist in einem Software-Entwurf eine Klasse, die das gesamte Processing als Monopol verwaltet, während andere Klassen primär die Daten kapseln.
- Im Klassendiagramm drückt sich dies dadurch aus, dass eine komplexe „Controller Klasse“ umgeben ist von zahlreichen einfachen Datenklassen.
- Dies hat den Effekt, dass fast die gesamte Verantwortung einer einzelnen Klasse überlassen wird.
- Im Grunde genommen entspricht der Blob einem prozeduralen Entwurf, der mit Hilfe einer objekt-orientierten Sprache implementiert wird.

The Blob  - ein Software Development AntiPattern

Symptome und Konsequenzen

- Einzelne Klasse mit einer großen Anzahl von Attributen, Operationen oder beidem.
  Mehr als 60 Attributen und Operationen => Blob.
- Eine Ansammlung der unterschiedlichsten Attribute und Operationen, die keine Beziehungen untereinander haben, sind in einer einzigen Klasse zusammengefasst.
- Eine einzige Controler-Klasse mit assoziierten, einfachen Daten-Objekt-Klassen.
- Fehlendes objekt-orientiertes Design. Die einzige Controler-Klasse kapselt oft die gesamte Funktionalität des Programmes, ähnlich einer prozeduralen main-function.
- Der Blob begrenzt die Möglichkeiten, das System zu modifizieren ohne die Funktionalität anderer, gekapselter Objekte zu beeinflussen.
- Typischerweise ist der Blob zu komplex für Wiederverwendbarkeit und Tests. Es ist nahezu ausgeschlossen, Untermengen der Funktionalität eines Blobs wiederzuverwenden.
- Es kann aufwendig sein, eine Blob-Klasse ins Memory zu laden, da sie exzessiv Resourcen verbraucht, selbst für einfache Operationen.

The Blob - ein Beispiel



The Blob - Lösungsansätze

Ziel einer Neustrukturierung ist

- Verhalten aus dem Blob herauszunehmen und gekapselten Datenobjekten zuzuordnen.

1. Identifiziere oder kategorisiere zusammengehörige Attribute und Operationen

- Die Zusammengehörigkeit sollte sich auf einen gemeinsamen Fokus, ein Verhalten oder eine Funktionalität im Gesamtsystem beziehen.
- Im Beispiel kapselt die Library-Klasse die Summe *aller* System-Funktionalitäten.
- Wir sammeln Operationen, die Bezug haben zum Katalog Management, wie Sort_Catalog und Search_Catalog.
- Ebenso identifizieren wir alle Operationen und Attribute die Bezug haben zu einem einzelnen Ausleiheobjekt, wie z.B. Print_Item, Delete_Item etc.

2. Finde eine geeignete Oberstruktur für diese neuen Gruppierungen („natural homes“) und sie dort einzubinden.

- Im Beispiel sind dies die bereits vorhandenen Klassen Catalog und Item.

The Blob - die überarbeitete Lösung

3. Löse alle redundanten oder indirekten Beziehungen und ersetze sie durch direkte Beziehungen

- Im Beispiel betrifft dies die Beziehung zwischen Item und Library:
  - Jedes Item steht zunächst in unmittelbarer Beziehung zu einem Katalog und nur über diesen - also indirekt - auch in Beziehung zur gesamten Bücherei.
  - Wir verzichten auf die Beziehung zwischen Item und Library und definieren die direkte Beziehung zwischen Catalog und Item.

4. Prüfe, ob assoziierte Klassen ggf. als Klassen erkannt werden können, die von einer gemeinsamen Basisklasse abgeleitet sind.

5. Ersetze zusammengehörige, transiente Beziehungen

- durch Klassen, die die entsprechenden Attribute und Operationen kapseln.
- Im Beispiel entstehen bei diesem Schritt Klassen wie
  - Checked_out_Item oder
  - Search_For_Item.

The Blob - die überarbeitete Lösung