Kapitel 12

Design Patterns

12.1 Das Role-Pattern

Faustregel: Klassenhierarchien nicht zu tief, denn 1. Nachbildungen „natürlicher“ tiefer Hierarchien sind meist nicht verhaltenskonformant; 2. Änderung des Objekttyps schwierig

Beispiel:

```
class Mitarbeiter {...}
class AussendienstMitarbeiter extends Mitarbeiter {...}
class InnendienstMitarbeiter extends Mitarbeiter {...}
```

typisches Subtyping (IS-A, $\subseteq$, Typkonformanz)

Problem: Aussendienstmitarbeiter kann zur Laufzeit nicht in Innendienst wechseln (man muß Clone-Objekt anlegen, Members kopieren, alle Referenzen auf Außendienstobjekte ändern!!)

⇒ oft sieht man Code mit dynamischen Tests:

```
class Mitarbeiter {
  ...
  switch (MitarbType) {
    case Aussendienst: ...
    case Innendienst: ...
  ...}
}
```

⇒ *nicht OO!!*

Lösung: dynamische Typisierung (s.d.) oder *Role-Pattern*

Mitarbeiter können eine *Rolle* spielen

Rollen können sich ändern, Mitarbeiter bleibt derselbe

⇒ Rollenklasse mit Unterklassen, Assoziation zwischen Mitarbeiter und Rolle

```
class Mitarbeiter {
    MitarbeiterRolle rolle;
    ...
}

class MitarbeiterRolle {
    Mitarbeiter mitarbeiter;
    ...
}

class AussendienstMitarbeiter extends MitarbeiterRolle {
  ... }

class InnendienstMitarbeiter extends MitarbeiterRolle {
  ... }
```

Vorteil: Softwaretechnisch werden eine Person und ihre Funktion in 2 Klassen getrennt (⇒ Kohäsion, Separation der Interessen)

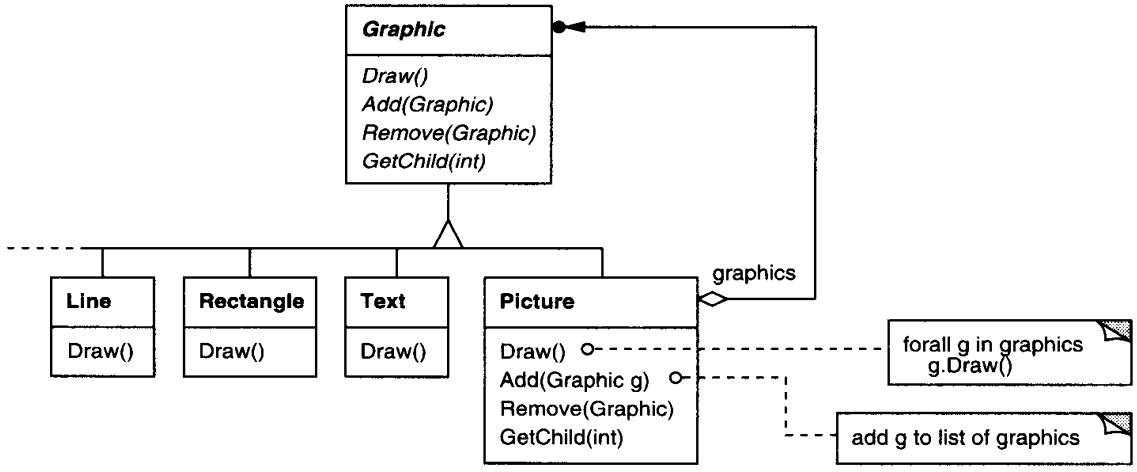
anderes Beispiel: Personen, Studenten, Professoren (Übung!)

12.2 Wiederholung: Composite

zusammengesetzte Objekte:

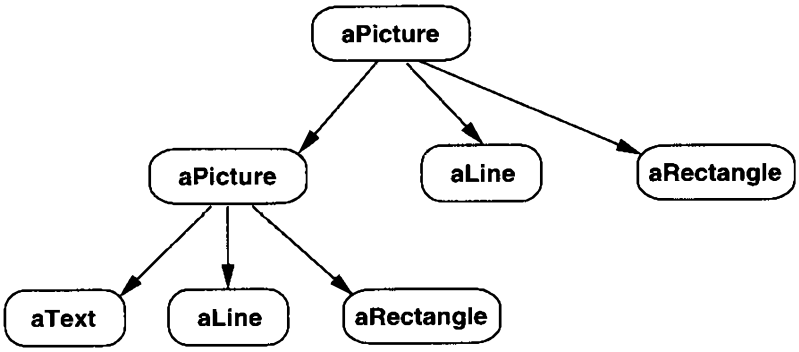
- verschiedene Arten "terminaler" Objekte ohne Unterkomponenten
- verschiedene Arten zusammengesetzter Objekte mit Unterkomponenten

konkretes Beispiel: graphische Objekte[1]

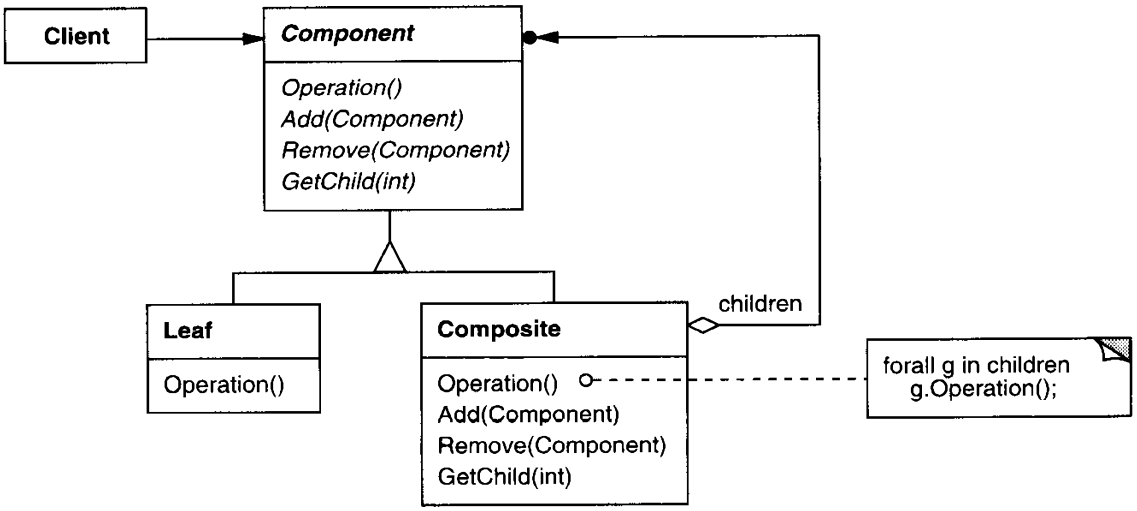


[1]Die folgenden Abbildungen sind aus Gamma et al., Design Patterns

Beispiel-Objektstruktur:



allgemeines Pattern als Klassendiagramm:

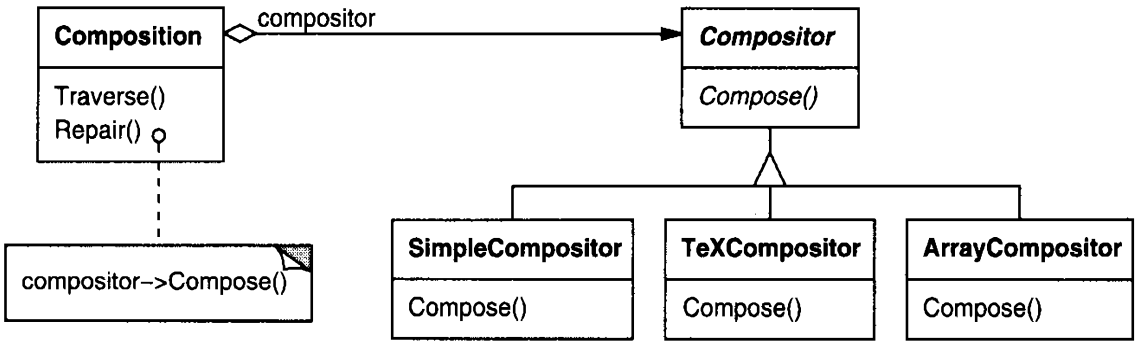


12.3 Wiederholung: Strategy

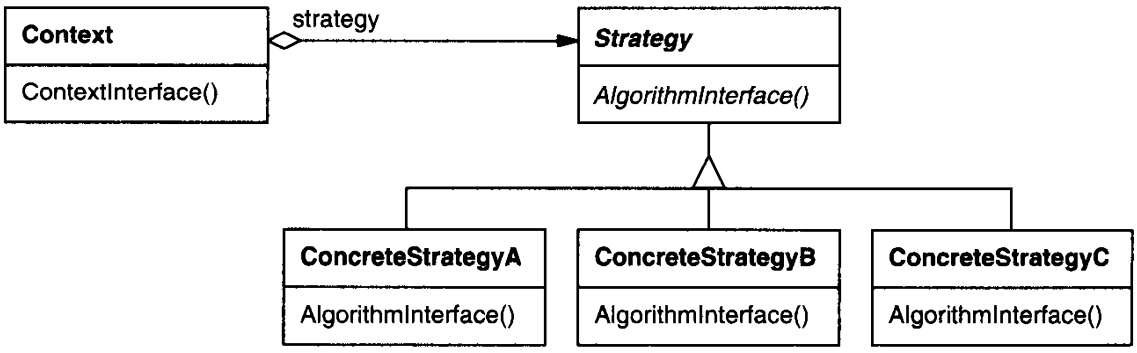
Abkapselung einer Familie von Algorithmen

einfaches Beispiel: Textanzeige mit Zeilenumbruch. Verschiedene Umbruchstrategien: Simple (zeilenweise), Tex (optimal für ganzen Absatz), Array (Tabelle)

“Composition” (Text) delegiert Zeilenumbruch an “Compositor”



allgemeines Muster:



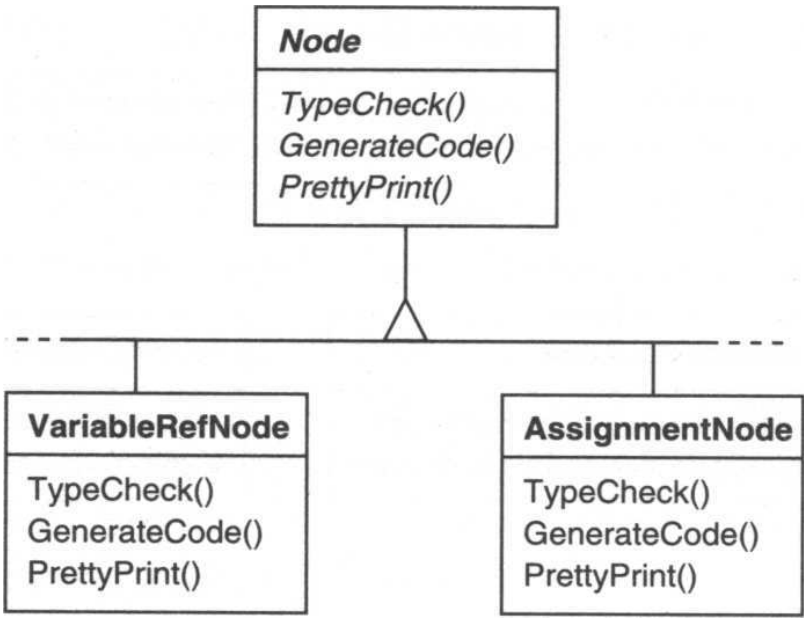
Anwendung auf MVC: Controller ≅ Compositor (vgl. “Handler”!)
Controller ruft wiederum Operationen des “Models” auf

Vorteil: Reaktion auf Events ist von View völlig entkoppelt und kann leicht ausgetauscht werden

12.4 Visitor

Gegeben: Knoten-Struktur gemäß Composite-Pattern (zB abstrakter Syntaxbaum) + Menge von Knoten-Operationen, die für jeden Knotentyp redefiniert sind

Beispiel:

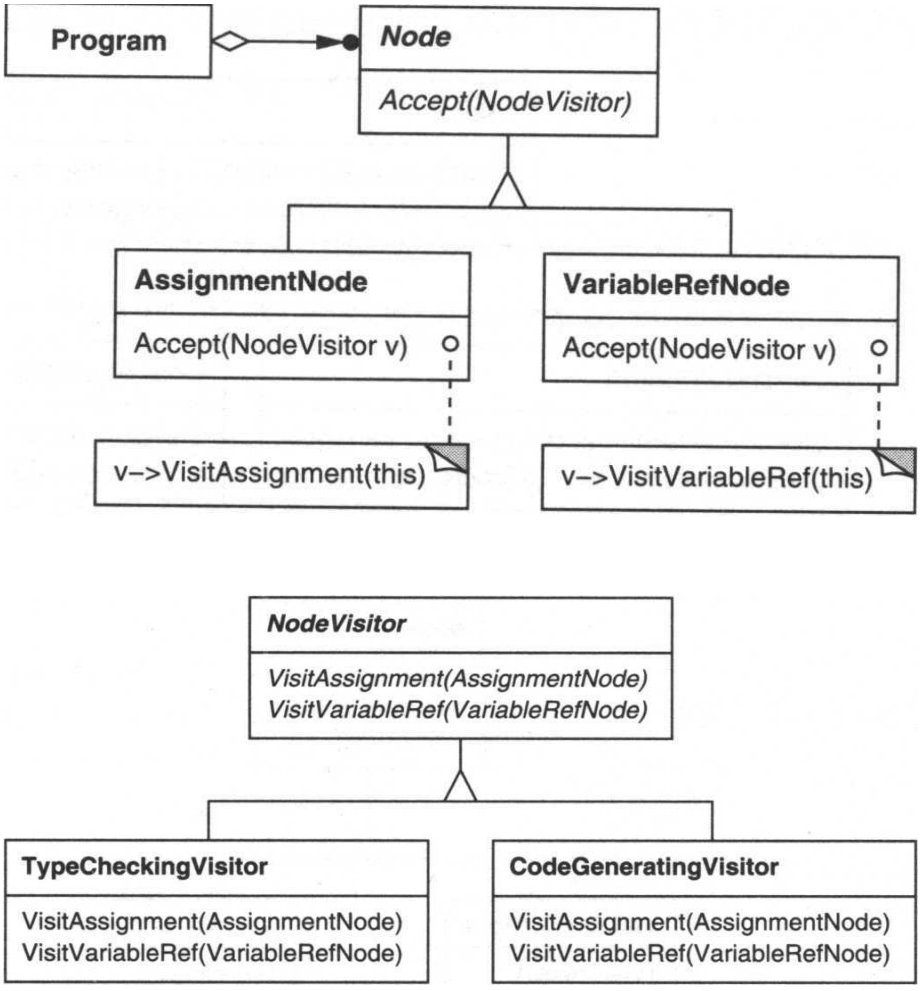


Nachteil: Klasse enthält Methoden verschiedenster Art, die jedoch orthogonal zur Knotenstruktur jeweils zusammengefasst werden können

⇒ Visitor: zwei orthogonale Hierarchien, eine für Knotenarten, eine für „Bearbeitungs-Aspekte“!

In jeder Klasse 1 Methode pro Knotenklasse; Knoten bekommen „accept“ Methode, die Visitorobjekt übergeben bekommt und damit die passende „Visit“ Methode aufruft.

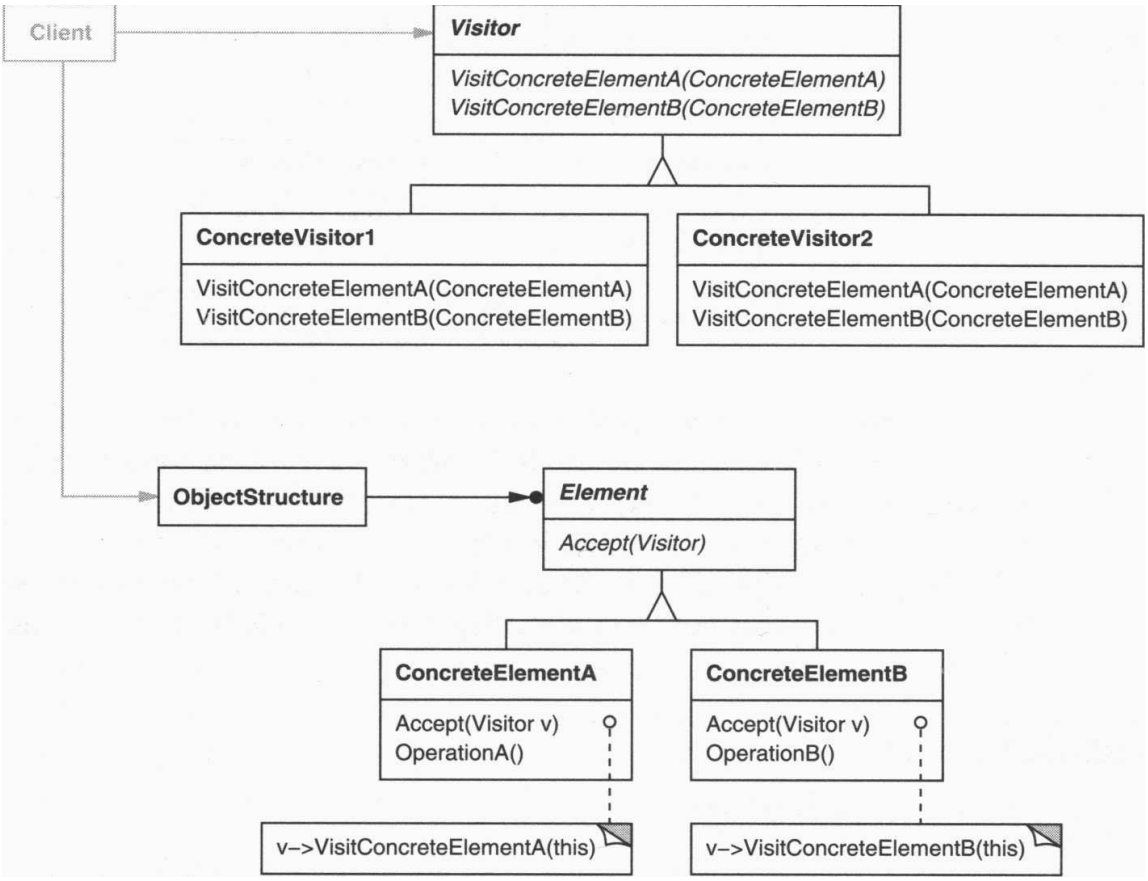
Zum gewünschten Bearbeitungsaspekt wird entsprechendes Visitor-Unterklassenobjekt erzeugt und an „accept“ übergeben; in diesem kann man auch Zwischergebnisse akkumulieren:



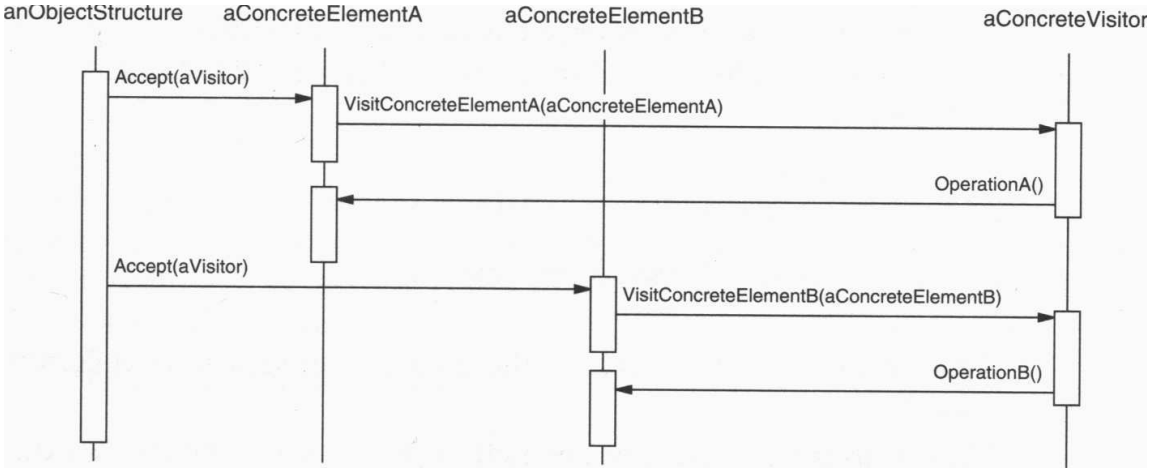
Beispielcode:

```
class AssignmentNode {
   ...
   void accept(NodeVisitor v) {
      v.visitAssignment(this); }
}

class TypeCheckingVisitor implements Visitor {
   void visitAssignment (AssignmentNode n) {
      l = n.getVar();
      r = n.getExpr();
      l.accept(this);  // this.visitVarRef(l);
      r.accept(this);  // this.visitExpr(r);
      ... Typcheck ...

   }
}

a = new AssignmentNode (new VarRef(), new Expr());
if (typcheck)
   v = new TypeCheckingVisitor();
else
   v = new CodeGeneratingVisitor();
a.accept(v);
```

allgemeines Schema:



Sequenzdiagramm:



Achtung: nur sinnvoll, wenn sich Knotenstruktur kaum ändert, Bearbeitungsaspekte jedoch häufig. Denn neue Knotenstruktur erfordert komplett neue Visitor-Interfaces; neuer Bearbeitungsaspekt nur 1 neue Visitor-Unterklasse

Bem: In Sprachen mit Multimethoden (s.o.) kann man auf Visitor verzichten (Übung!)

12.5 Factory

Gegeben: Familie von isomorphen Hierarchien (zB Widget-Familie für verschiedene Fenstersysteme)

Unterklassen sind oft nicht verhaltenskonformant ⇒ Klient muss passendes Unterklassenobjekt (i.e. Widget für spezifisches Fenstersystem) selbst erzeugen; alle diese Unterklassenobjekte müssen zusammenpassen (selbe Klasse); Klient muss viele konsistente Fallunterscheidungen machen

Beispiel: Versuch 1

```
class MSWidget { ...}

class MSMenu extends MSWidget {...}

class MSScrollbar extends MSWidget {...}

class XWidget {...}

class XScrollbar extends XWidget {...}

class XMenu extends XWidget {...}
```

Nachteil: Klient muss bei jeder GUI-Aktion Fallunterscheidung nach Fenstersystem machen

Versuch 2: „Invertierung“

```
class Scrollbar { ...}

class MSScrollbar extends Scrollbar {...}

class XScrollbar extends Scrollbar {...}

class Menu {...}

class MSMenu extends Menu {...}

class XMenu extends Menu {...}
```

besser, denn es werden dynamisch viel weniger Fallunterscheidungen gemacht (nur bei Widgeterzeugung)

```
if (...)
    w = new MSMenu();
else
    w = new XMenu();
```

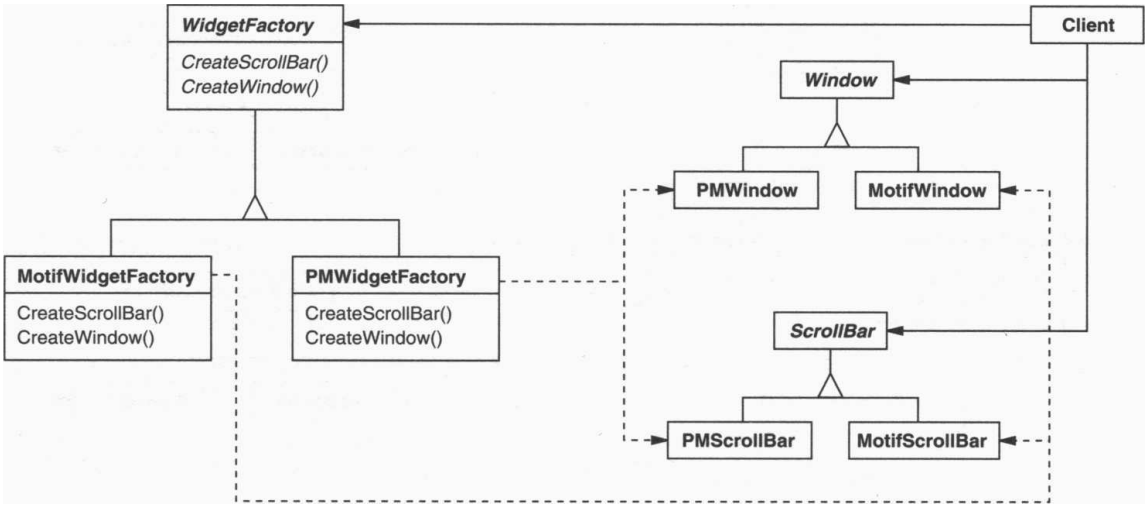
den Rest macht die dynamische Bindung

⇒ Factory: eine weitere isomorphe Hierarchie für *Konstruktoren*:

⇒ noch besser: man wird auch nicht die Fallunterscheidungen bei Objekterzeugung los

Jede Klasse der Factory-Hierarchie enthält alle Konstruktoren der Familie und bietet sie als `create...` an

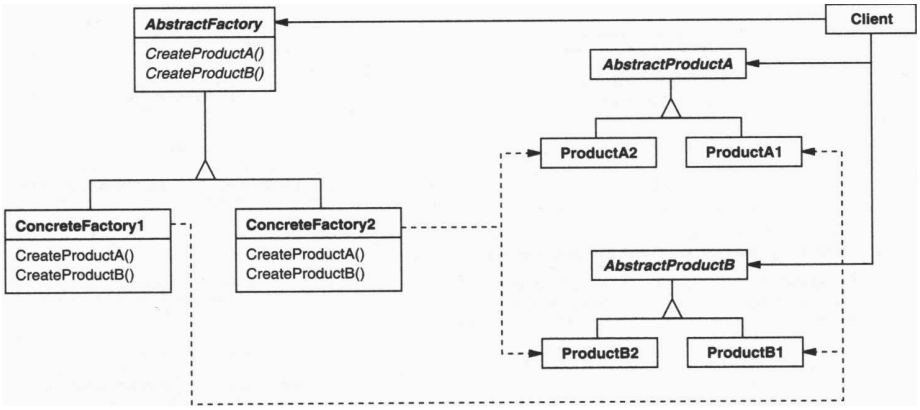
Beispiel:



⇒ Fallunterscheidung (zB bez. Fenstersystem) nur noch bei Erzeugung des Factory-Objektes! Alle anderen Klienten erzeugen Widget-Objekte mittels `create...`, den sog. Factory-Methods

```
WidgetFactory myFactory;
switch (WinType) {
  case Motif: {
        myFactory = new MotifWidgetFactory();}
  case PresMang: {
        myFactory = new PMWidgettFactory();}
}

myWindow = myFactory.createWindow();
myScrollbar = myFactory.createScrollbar();
```

Entspricht dem Grundprinzip, Fallunterscheidung durch Vererbung zu ersetzen! Allgemeines Schema:



Achtung: neue Fenstersysteme einbringen ist einfach (1 neue Factory-Unterklasse), aber neue Widgets („Familienmitglieder“) einzuführen erzwingt globale Änderung in der Factory

12.6 Abschlussbemerkung zu Design Patterns

Viele Patterns behandeln Situationen, wo zwei orthogonale Varianten-Dimensionen „aufeinanderstoßen“:

- Strategy: Knotentypen/Algorithmenvarianten;
- Observer: verschiedene Objekte/verschiedene Beobachter;
- Visitor: Knotentypen/Bearbeitungsaspekte;
- Factory: Produktfamilie/Plattformhierarchie;
- Bridge: verschiedene Objektarten/verschiedene Implementierungsarten.

Dies lässt sich durch nur eine Klassenhierarchie prinzipiell schlecht ausdrücken („Tyrannei der dominanten Zerlegung“).

Patterns bieten spezielle Lösungen, in denen immer wieder die Struktur der „Brücke“ auftaucht

Ein allgemeiner Ansatz sind *Aspekte* (s.u.)