Entwurfsmuster (Design Patterns)

Entwurfsmuster (Design Patterns)

In der alltäglichen Programmierarbeit tauchen viele Probleme auf, die man schon einmal gelöst hat und die man in der Zukunft wieder lösen müssen wird.

Entwurfsmuster sind Dokumentation von Lösungs- und Designansätzen.

- Wiederverwendung von erfolgreichen Entwurfsansätzen
- Dokumentation von Lektionen, die bei der Lösung von realen Problemen gelernt wurden
- Kommunikation
  - Entwurfsmuster haben Namen und erleichtern so die Beschreibung von Struktur und Funktionsweise von Programmen
  - fremde Programme, die einem bekannten Entwurfsmuster entsprechen, sind leichter lesbar

Was ist ein Entwurfsmuster?

Entwurfsmuster beschreiben Wissen über objektorientiertes Design.

Ein Entwurfsmuster

- löst ein bestimmtes Problem
- beschreibt ein *erprobtes* Lösungskonzept
- ist nicht direkt offensichtlich (es wäre nicht sinnvoll eine ohnehin offensichtliche Lösung zu beschreiben)
- erfasst das Zusammenspiel von Komponenten (das sich im Allgemeinen nicht direkt in der Programmiersprache ausdrücken lässt)

Entwurfsmuster

Ein Entwurfsmuster besteht aus folgenden Teilen:

Name	wichtig für Kommunikation
Problem	Beschreibung der Situation, in der das Muster angewendet werden kann, und deren Kontext.
Lösung	informelle Beschreibung der Komponenten des Designs, ihrer Beziehungen, Aufgaben und Pflichten sowie ihrer Kommunikation.
Beispiele	Illustration der Lösung an repräsentativen Beispielen
Evaluation	Diskussion von Vor- und Nachteilen des Entwurfsansatzes

Entwurfsmuster

Strukturmuster beschreiben die Zusammensetzung von Objekten und Klassen zu größeren Strukturen.

Beispiele: *Composite*, Adapter, Decorator, Flyweight, Proxy, ...

Verhaltensmuster beschreiben Möglichkeiten zur objektorientierten Implementierung von Algorithmen und zur Interaktion zwischen Objekten und Klassen.

Beispiele: Iterator, Observer, *Visitor*, Interpreter, ...

Erzeugungsmuster beschreiben Möglichkeiten zur Objekterzeugung.

Beispiele: *Abstract Factory*, Builder, Singleton, ...

...

Strukturmuster – Composite

Problem:

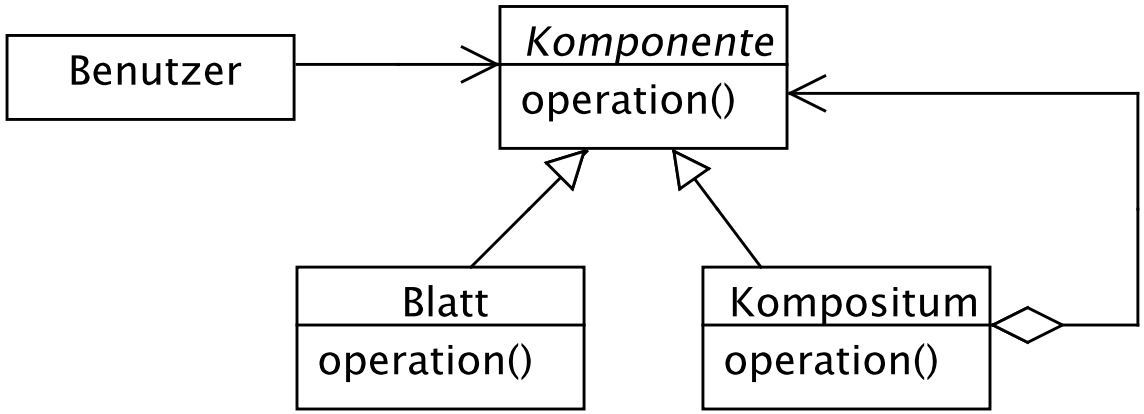
- Repräsentiere eine Menge von Objekten, in der einzelne Objekte aus anderen Objekten in der Menge zusammengesetzt sein können.
- Zusammengesetze Objekte sollen wie einzelne Objekte behandelt werden können.

Beispiele:

- Eine geometrische Figur ist entweder
  - eine Linie,
  - ein Kreis,
  - ein aus mehreren geometrischen Figuren bestehendes Bild.
- Ein arithmetischer Ausdruck ist entweder
  - eine konstante Zahl,
  - eine Variable,
  - eine Verknüpfung zweier arithmetischer Ausdrücke mit einer Binäroperation.

Strukturmuster – Composite

Das Entwurfmuster Composite beinhaltet folgende Repräsentation:



Komponente:

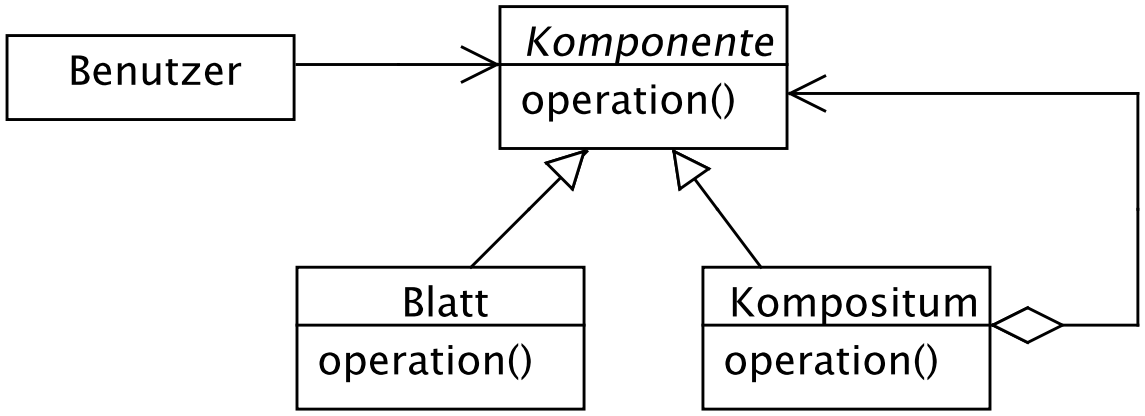
- Interface für die Objekte in der Komposition (z.B. Auswerten bei arithmetischen Ausdrücken)
- Interface für gemeinsames Verhalten (z.B. Verschieben bei geometrischen Figuren)
- Interface für den Zugriff auf die Teilkomponenten.

Blatt: primitives Objekt

Kompositum: zusammengesetztes Objekt

Benutzer: interagiert nur mit der Komponente

Strukturmuster – Composite



Beispiel: arithmetische Ausdrücke in AntBrain

Komponente	IExp
Blatt	IExpConst, IExpVar
Kompositum	IExpBinOp
operation	eval

Strukturmuster – Composite

Evaluation

- definiert Teil-Ganzes-Hierarchien von primitiven und zusammengesetzten Objekten
- Benutzung dieser Hierarchien ist einfach. Zusammengesetzte Objekte können wie primitive Objekte benutzt werden.
- Hierarchie leicht durch neue Objekte erweiterbar, ohne dass Benutzer ihren Code ändern müssen
- Benutzer kann keine neue Operation selbst definieren und zur Hierarchie hinzufügen (Beispiel: `IExp.getFreeVariables()`)
- Quelltext der `operation()` über mehrere Klassen verstreut (Beispiel: `IExp.eval()`).

Verhaltensmuster – Visitor

Probleme der Datenrepräsentation eines Composite-Patterns:

- Benutzer kann keine neue Operation selbst definieren und zur Hierarchie hinzufügen.
- Quelltext der `operation()` über mehrere Klassen verstreut.

Das Visitor-Pattern löst diese Probleme:

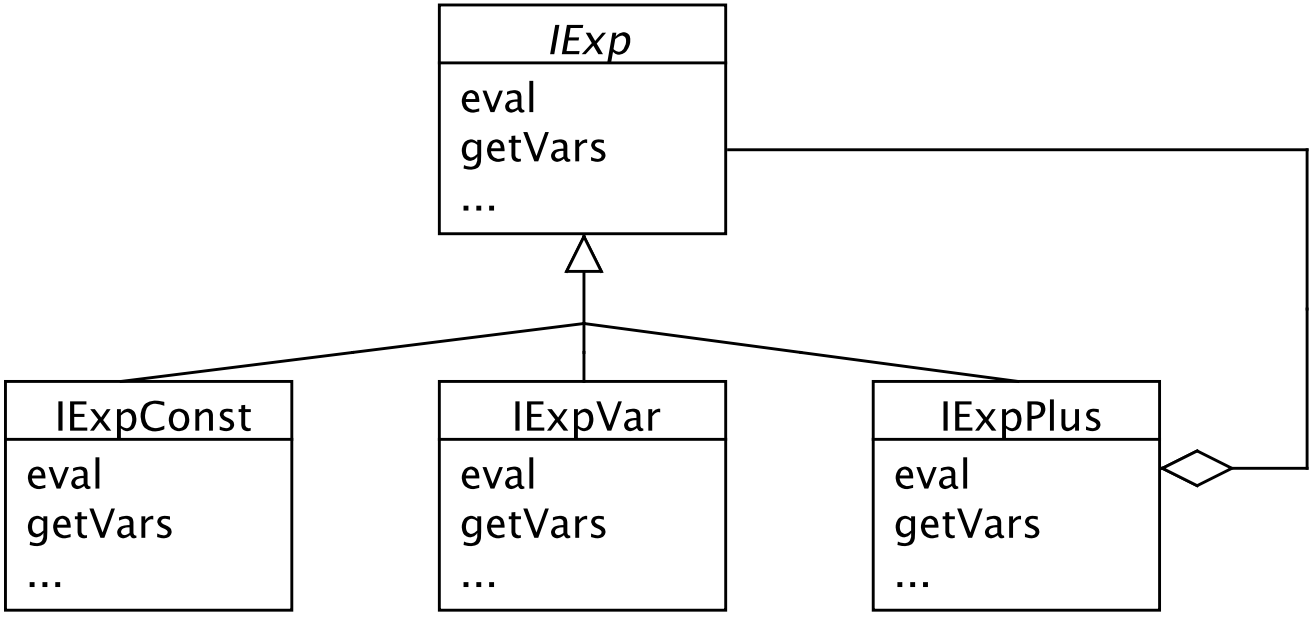
- Es erlaubt, neue Operationen zu definieren, ohne die Klassen der Struktur zu ändern.
- Es erlaubt, Operationen, die auf einer Objektstruktur ausgeführt werden sollen, kompakt zu repräsentieren.

Verhaltensmuster – Visitor

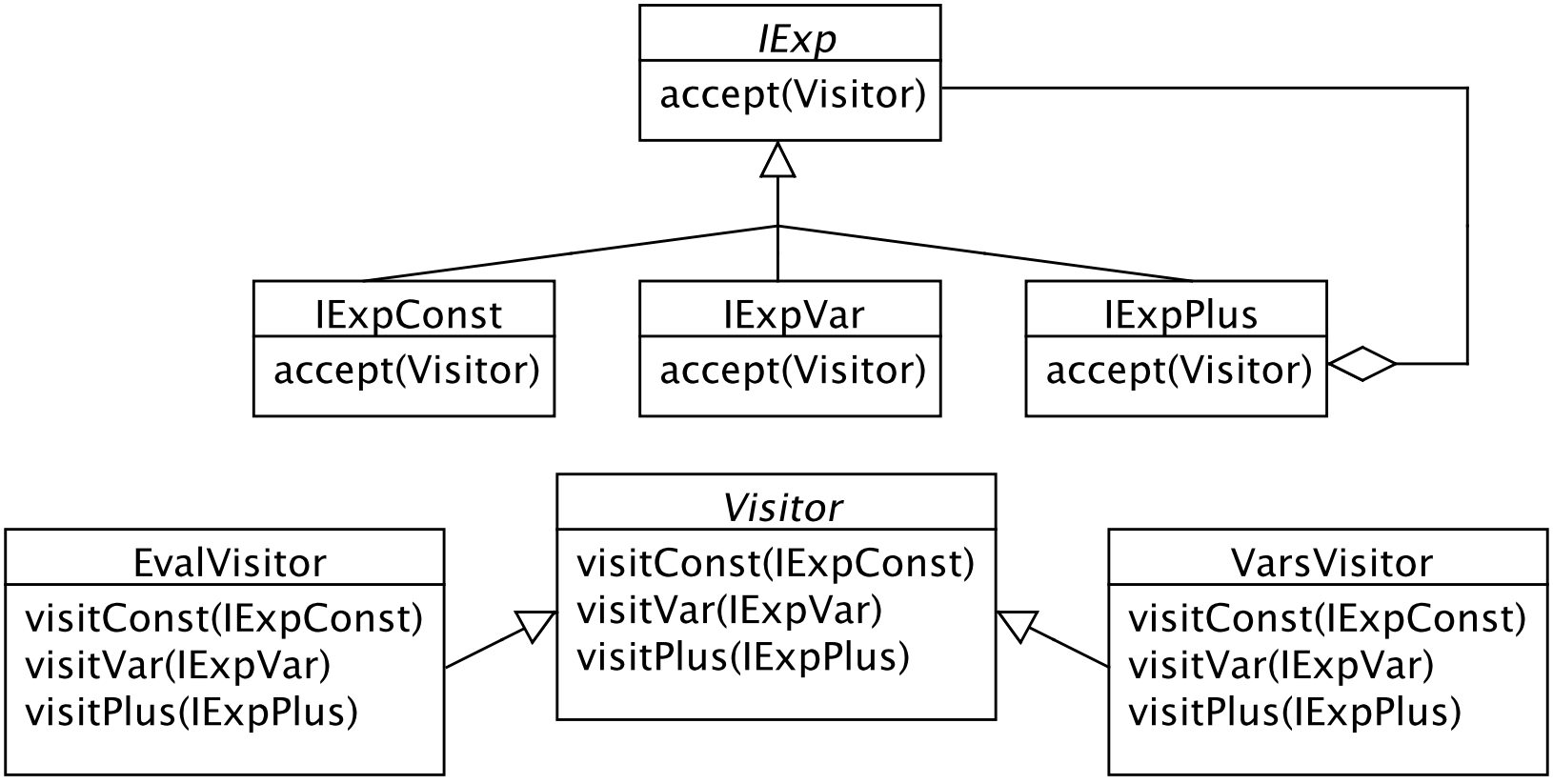
Idee:

- Sammle die Definitionen der Operationen auf der Objektstruktur in einem Visitor-Objekt.
- Ersetze die verschiedenen Operationen durch eine einzige `accept`-Methode.

Beispiel



Verhaltensmuster – Visitor



`accept(v)` in `IExpConst` durch `v.visitConst(this)` gegeben, usw.

$\Rightarrow$ Auslagerung der Methodendefinitionen in Visitor-Objekte.

Verhaltensmuster – Visitor

```
abstract class IExp {
  abstract <T> T accept(IExpVisitor<T> v);
}

class IExpVar extends IExp {
  String varName;
  <T> T accept(IExpVisitor<T> v) { return v.visitVar(this); }
}

class IExpConst extends IExp {
  Integer value;
  <T> T accept(IExpVisitor<T> v) { return v.visitConst(this); }
}

class IExpPlus extends IExp {
  IExp e1, e2;
  <T> T accept(IExpVisitor<T> v) { return v.visitPlus(this); }
}
```

Verhaltensmuster – Visitor

```
abstract class IExpVisitor<T> {
    abstract T visitVar(IExpVar v);
    abstract T visitConst(IExpConst c);
    abstract T visitPlus(IExpPlus p);
}
```

Funktionen für arithmetische Ausdrücke können nun in zentral in einem `IExpVisitor`-Objekt aufgeschrieben werden.

Beispiele:

- Auswertung eines Ausdrucks: `EvalVisitor`
- Variablen in einem Ausdruck: `VarsVisitor`

Verhaltensmuster – Visitor

Auswertung eines Ausdrucks `iexp`:

```
iexp.accept(new EvalVisitor(memory));

class EvalVisitor extends IExpVisitor<Integer> {
    Map<String, Integer> memory;

    EvalVisitor(Map<String, Integer> memory) {
      this.memory = memory;
    }
    Integer visitVar(IExpVar v) {
      return memory.get(v.varName);
    }
    Integer visitConst(IExpConst c) {
      return c.value;
    }
    Integer visitPlus(IExpPlus p) {
      return p.e1.accept(this) + p.e2.accept(this);
    }
}
```

Verhaltensmuster – Visitor

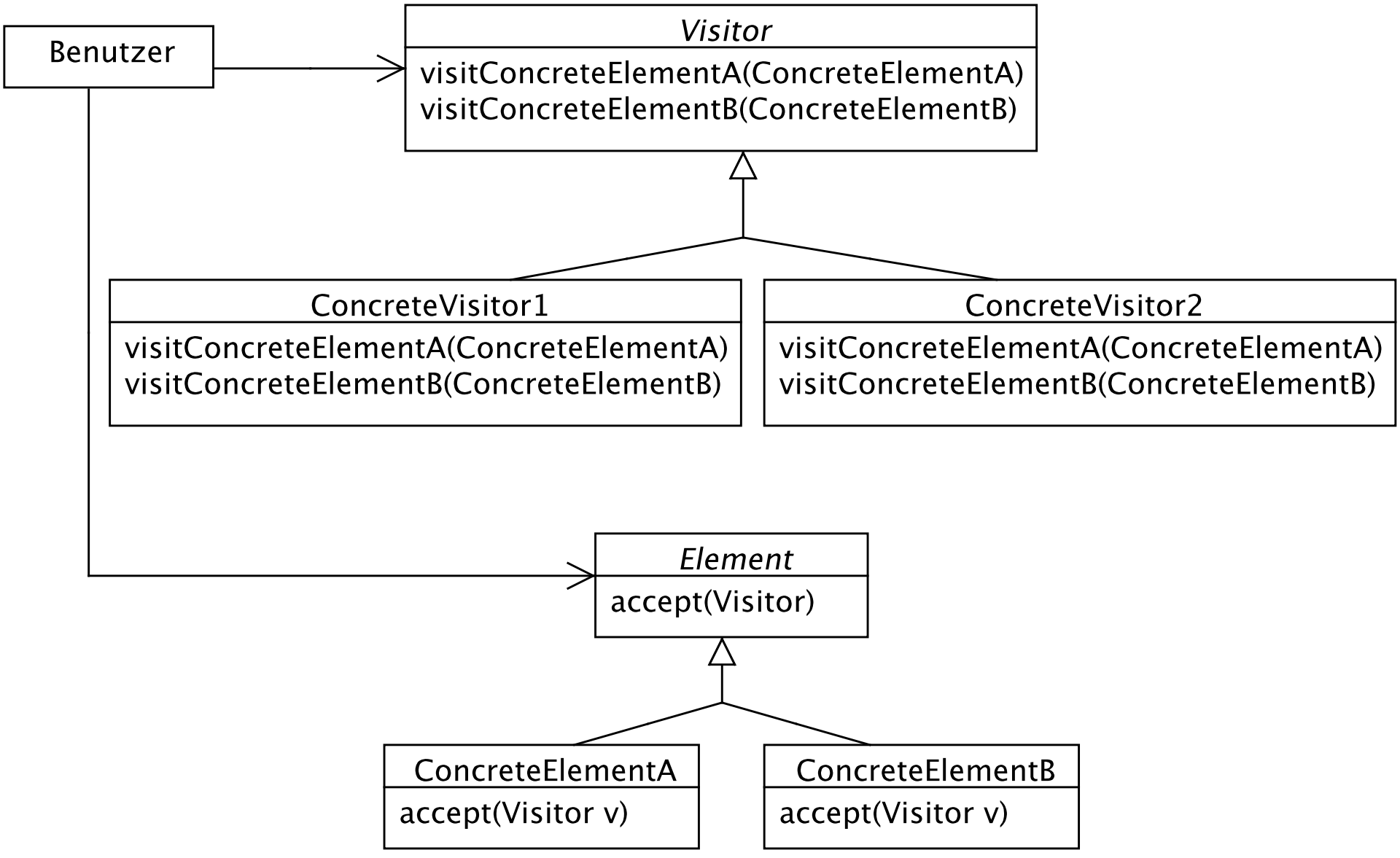
Berechnung der Variablenmenge eines Ausdrucks `iexp`:

```
iexp.accept(new VarsVisitor());
```

```
class VarsVisitor extends IExpVisitor<Set<String>> {

    Set<String> visitVar(IExpVar v) {
      return java.util.Collections.singleton(v.varName);
    }
    Set<String> visitConst(IExpConst c) {
      return java.util.Collections.emptySet();
    }
    Set<String> visitPlus(IExpPlus p) {
      Set<String> s = new HashSet<String>();
      s.addAll(p.e1.accept(this));
      s.addAll(p.e2.accept(this));
      return s;
    }
}
```

Verhaltensmuster – Visitor



Verhaltensmuster – Visitor

Evaluation

- Es ist leicht, neue Operation zu definieren, ohne die Klassen in der Hierarchie ändern zu müssen.
- Die vielen Fälle einer einzigen Operation werden in einem Visitor zusammengefasst. (Vergleiche: In der Klassenhierarchie waren die verschiedenen Fälle verschiedener Operationen zusammengefasst)
- Hinzufügen einer neuen Klasse zur Hierarchie ist schwierig, da man auch alle Visitor-Klassen anpassen muss.
- Zur Definition der einzelnen Funktionen in einem Visitor-Objekt braucht man Zugriff auf den Zustand der Objekte in der Objekthierarchie.
  $\Rightarrow$ Kapselung nicht immer möglich.

Erzeugungsmuster – Abstract Factory

Konstruktoren sind nicht immer die beste Möglichkeit Objekte zu erzeugen.

Beispiel: Abstraktion von Systemtreibern

- Portabilität durch Abstraktion von Systemeigenschaften (Beispiel: Austausch von Datenbanksystemen)
- Programm muss auf Treiber zurückgreifen und Treiberobjekte erzeugen
- je nach System müssen verschiedene Treiber erzeugt werden (Beispiel: SQL Server oder PostgreSQL)

Erzeugungsmuster – Abstract Factory

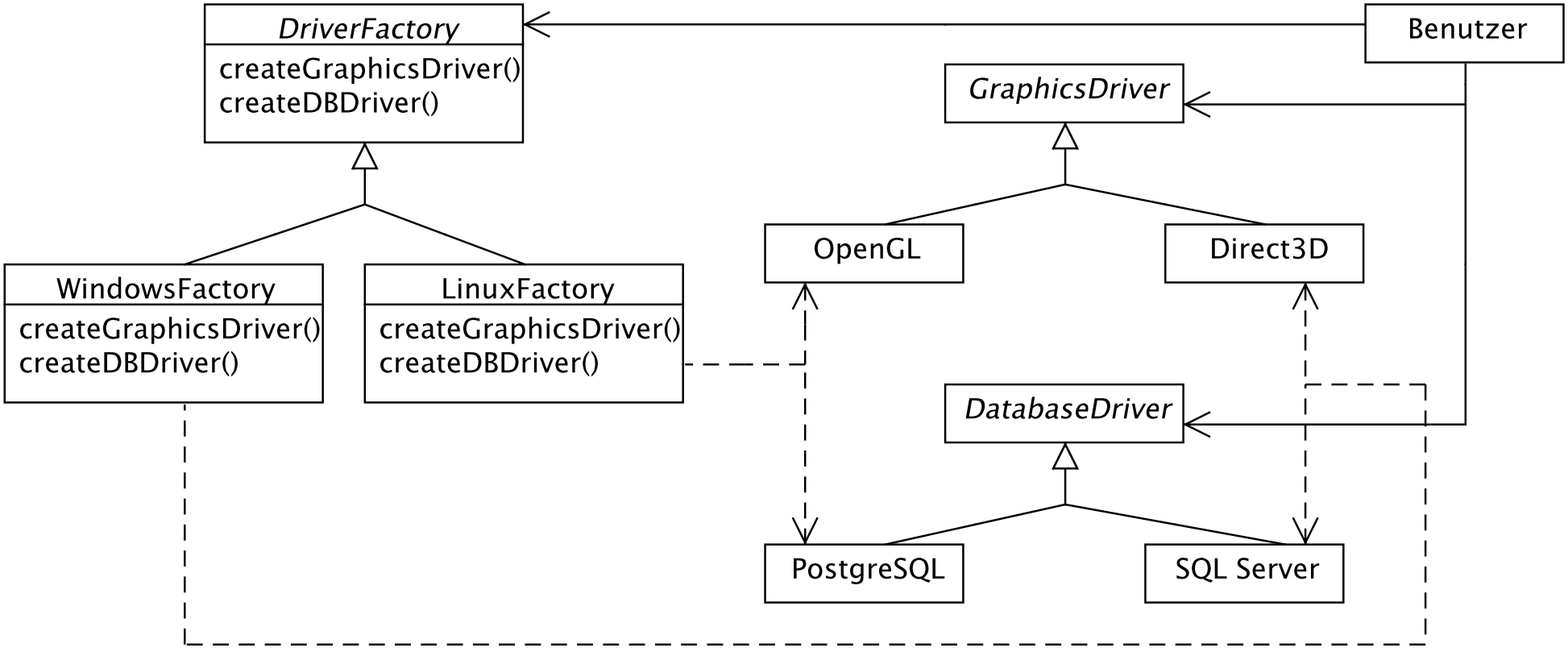
Abstract Factory

Schnittstelle zur Erzeugung von Familien zusammenhängender oder verwandter Objekte (z.B. Treiber), ohne dass dazu die konkreten Objektklassen bekannt sein müssen.

Idee:
„Erzeuge einen Datenbanktreiber.“
statt
„Erzeuge einen Treiber für PostgreSQL.“

Erzeugungsmuster – Abstract Factory

Struktur:



Erzeugungsmuster – Abstract Factory

`createGraphicsDriver()` und `createDBDriver()` sind Beispiele für *Factory-Methoden*

Factory-Methoden können in bestimmten Situation besser geeignet sein als Konstruktoren:

- Wiederverwendung von Objekten

  ```
  public static Boolean valueOf(boolean b);
  ```

- Erzeugung von geeigneten Unterklassen. Beispiel:

  ```
  public static <T> Set<T> synchronizedSet(Set<T> s);
  ```

  in `java.util.Collections` macht Menge thread-sicher
  - kann in zukünftigen Java-Versionen effizientere Implementierungen zurückliefern, ohne dass Benutzer ihren Code ändern müssten
  - Klasse, welche die Synchronisierung konkret implementiert, kann in `java.util` privat bleiben