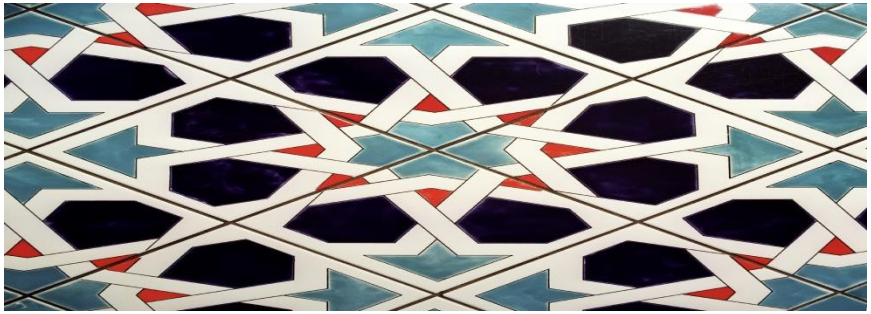
Programmieren 3

Kapitel 3: Entwurfsmuster

- Motivation
- Komposition
- Beobachter
- Singleton
- Proxy, Adapter
- Fliegengewicht
- Strategie
- Besucher

Martin Binder, FH Rosenheim, © 2015

Was ist ein Muster ?



„Ein Software-Architektur-Muster beschreibt ein bestimmtes, in einem speziellen Entwurfskontext häufig auftretendes Entwurfsproblem und präsentiert ein erprobtes generisches Schema zu seiner Lösung. Dieses Lösungsschema spezifiziert die beteiligten Komponenten, ihre jeweiligen Zuständigkeiten, ihre Beziehungen untereinander und die Art und Weise, wie sie kooperieren.“
Frank Buschmann et al., 1996

Motivation für Entwurfsmuster

- Gemeinsame Mustervokabulare sind mächtig
- Mit Muster können Sie mit weniger mehr sagen
- Auf Musterebene zu bleiben ermöglicht Ihnen länger im Entwurf zu bleiben
- Gemeinsame Vokabulare können ihrem Entwicklerteam Beine machen
- Gemeinsame Vokabulare können Junior-Entwickler dazu bringen, dass sie sich weiterentwickeln
- Entwurfsmuster wenden die wesentlichen OO-Prinzipien an, Sie vertiefen und verfestigen damit Ihre Fähigkeiten in der OOP

OO-Paradigmen und Muster

- Muster „funktionieren“ durch Kombination diverser OO-Paradigmen
- wichtigste Paradigmen: Schnittstellen, Vererbung, Objektkomposition, Delegation, Kapselung
- Beispiele für Einsatz in Muster:
  - **Strategie** verwenden Schnittstellen und Vererbung
  - **Proxy** und **Decorator** verwenden Delegation
  - **Komposition** verwendet Vererbung und Objektkomposition
  - **Singleton** verwendet Kapselung

OO-Prinzipien und Muster (1)

- Kapseln Sie das was variiert.
  → Factory

- Ziehen Sie die Komposition der Vererbung vor.
  → Adapter

- Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
  → Strategie, Adapter, Composite

- Streben Sie für Objekte die interagieren nach Entwürfen mit lockerer Bindung.
  → Observer

OO-Prinzipien und Muster (2)

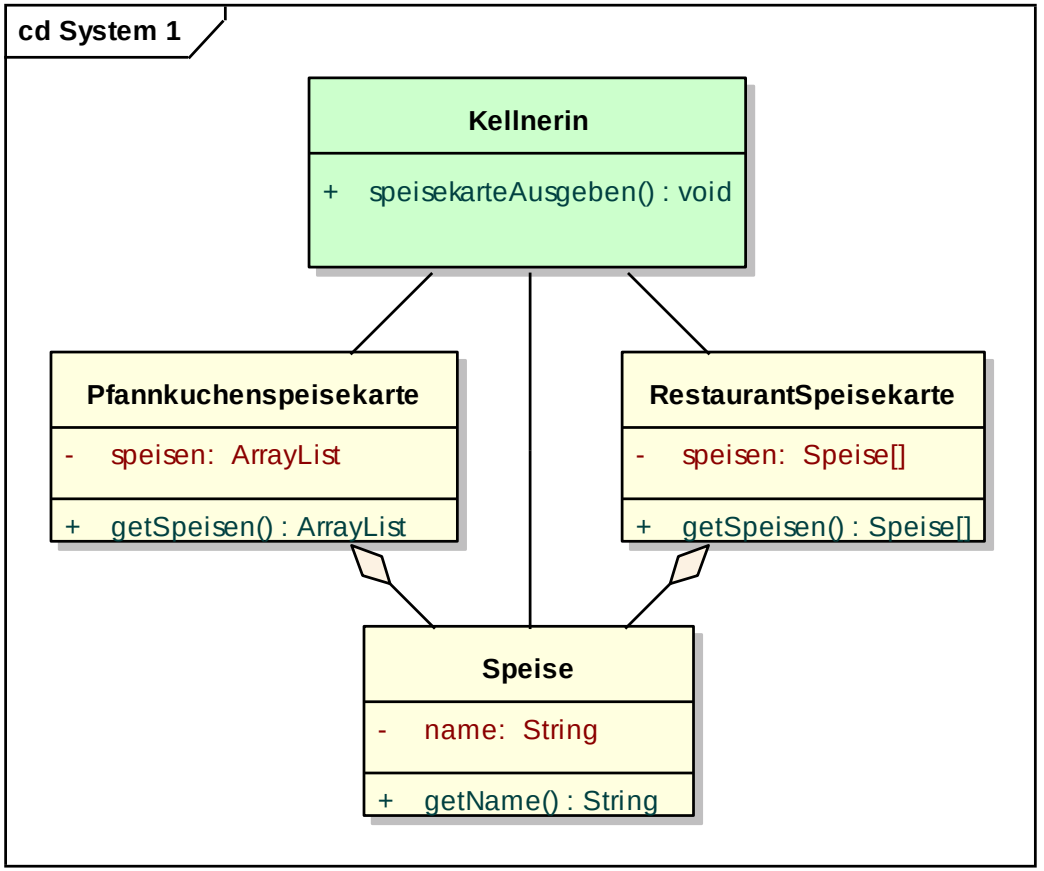
- Sprechen Sie nur mit Ihren Freunden.
  → Adapter, Fassade
- Versuchen Sie nicht uns anzurufen, wir rufen Sie an.
  → Observer, Factory
- Stützen Sie sich auf Abstraktionen. Stützen Sie sich nicht auf konkrete Klassen.
  → Factory
- Weitere OO-Prinzipien, die sich auch in den Patterns wiederfinden:
  - Eine Klasse sollte nur einen Grund haben sich zu ändern.
  - Klassen sollten für Erweiterungen offen, aber für Veränderungen geschlossen bleiben

Komposition

- Anwendung
  - Für Hierarchie von Objekten
  - Einfache rekursive Traversierung von Objektstrukturen
  - Client kann Unterschiede zwischen einzelnen und zusammengesetzten Objekten ignorieren
  - Beispiele: Grafikbehälter, Dateisystem mit Verzeichnissen und Dateien, geometrische Figuren
- Implementierung
  - Gemeinsame Oberklasse für Primitive und Kompositum
  - Kompositum leitet Operationen an Kinder weiter
  - Ein Iterator bietet eine allgemeine Schnittstelle für die Durchquerung eines Aggregats ohne seine interne Struktur zu kennen

Beispiel Restaurant

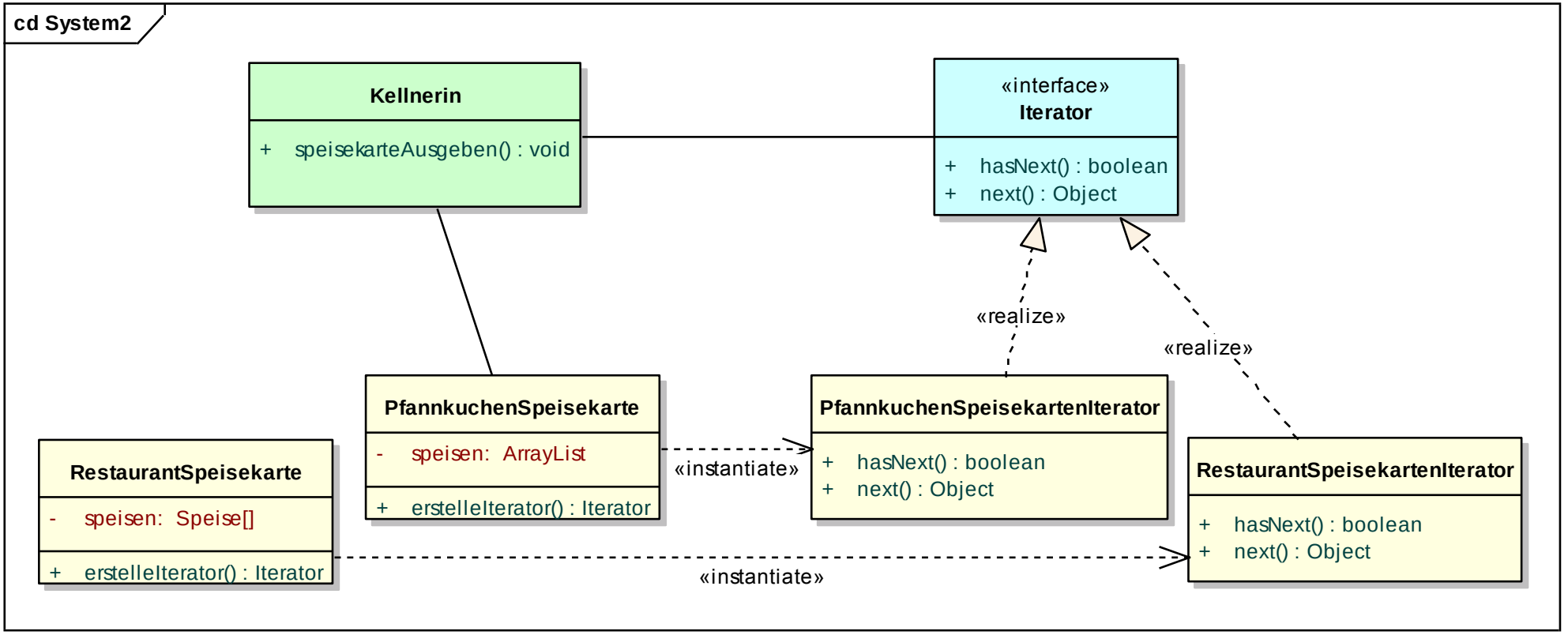
- Problem
  - zwei Restaurants mit eigenen Speisekarten fusionieren
  - Die Implementierung der Speisekarten ist bereits gegeben
  - Kellnerin braucht eine Methode um alle Speisen auszugeben
- Lösung 1: Direkte Assoziation
  - Zwei Speisekartenklassen mit unterschiedlicher interner Speicherung
  - Kellnerin muss Implementierung berücksichtigen



(aus „Entwurfsmuster von Kopf bis Fuß“, Freeman & Freeman, O‘Reilly)

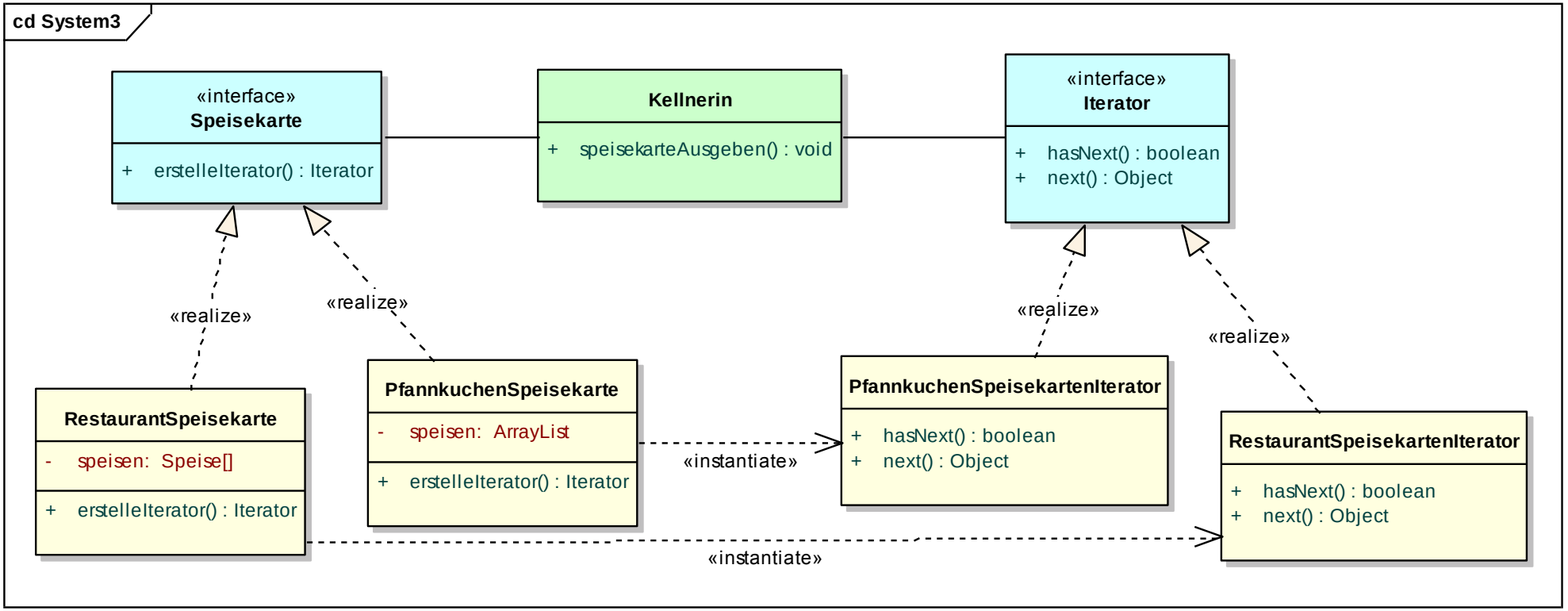
Restaurant Lösungsansatz mit Iterator-Muster

- Lösungsansatz 2: Iterator-Muster
  - beide Speisekartenklassen liefern einen Iterator der Speisekarte durchläuft
  - Kellnerin wird von der Implementierung der Speisekarten entkoppelt

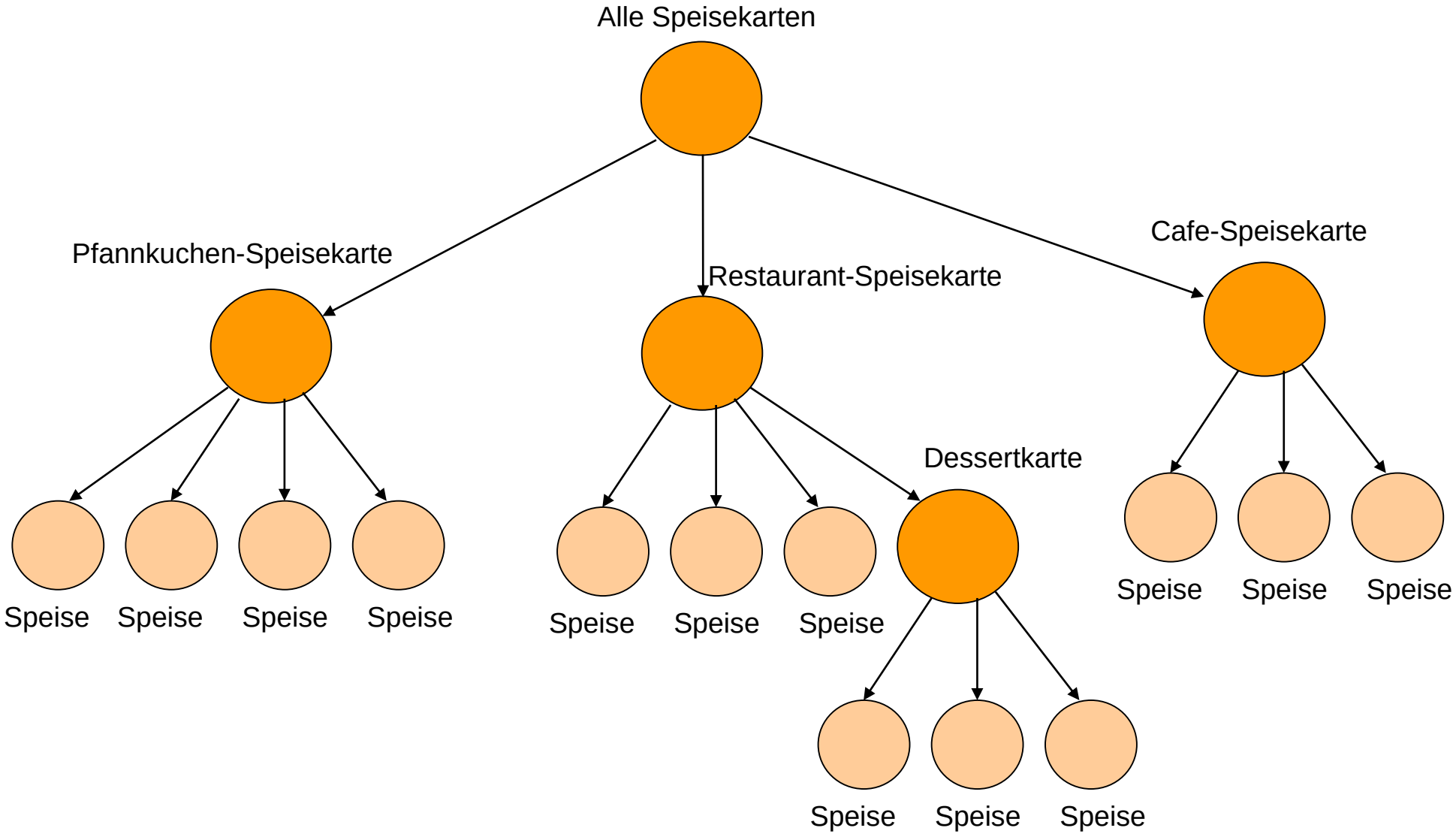


Restaurant mit Iterator und weiterer Schnittstelle

- Lösungsansatz 3: Iterator und Speisekarte-Interface
  - Beide Speisekarten implementieren ein gemeinsames Interface
  - Kellnerin kennt nur noch Schnittstelle und muss nicht gegen Implementierungen programmieren

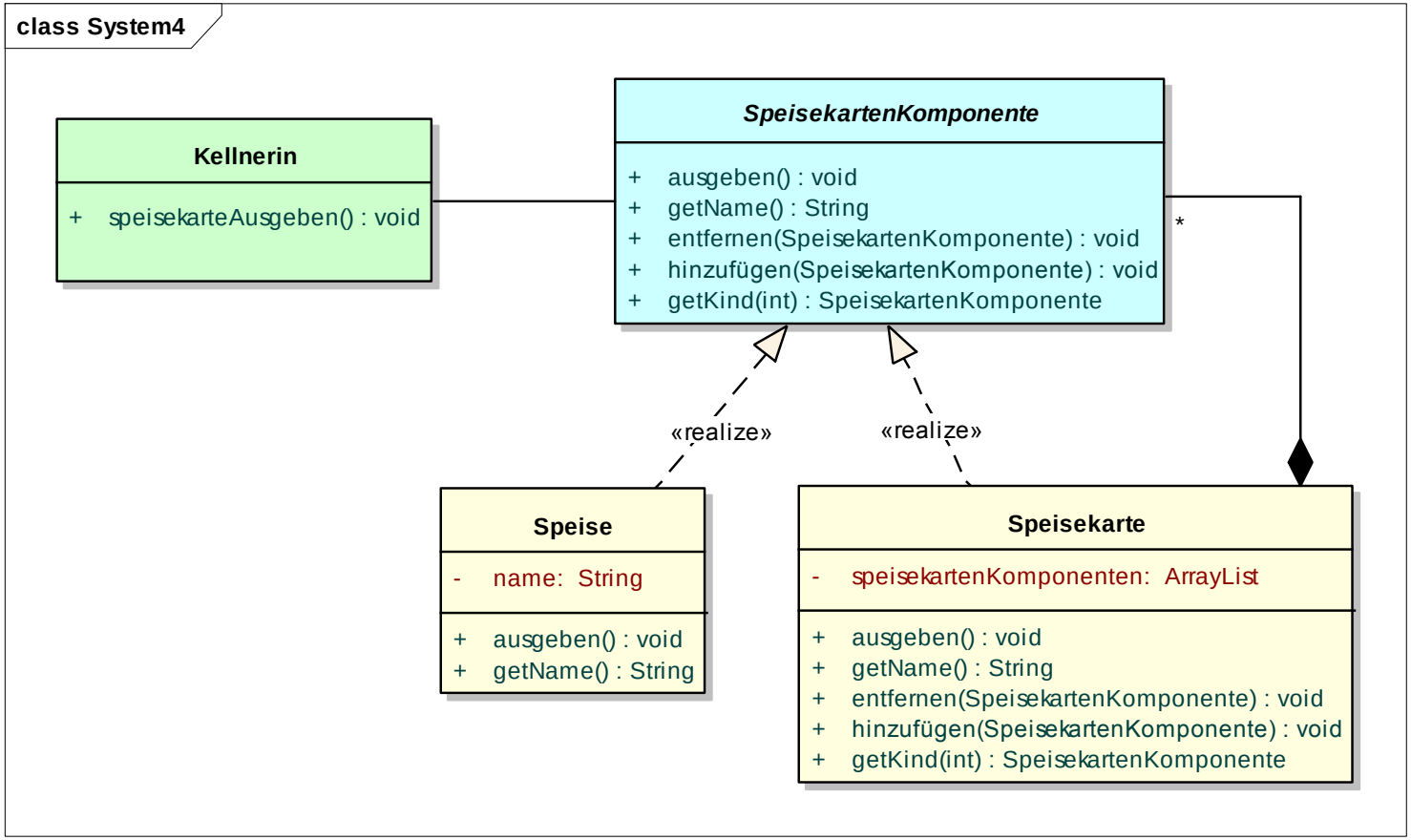


Zusatzanforderung:
Speisekarte kann auch Speisekarten enthalten



Restaurant mit Composite-Muster

- Lösungsansatz 4: Komposition
  - Speisekarte besteht aus Speisen und Speisekarten
  - Methode ausgeben( ) iteriert über Elemente der Speisekarte und durchläuft rekursiv die Baumstruktur

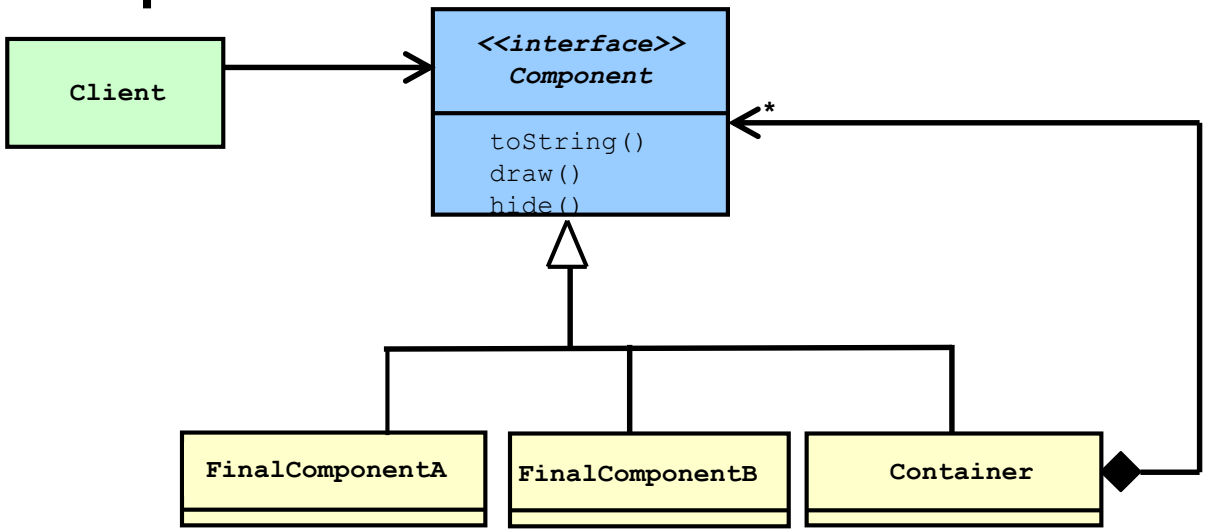


Rekursive Behälter und Komposition

1. Jedes Element ist entweder einfach oder zusammengesetzt.

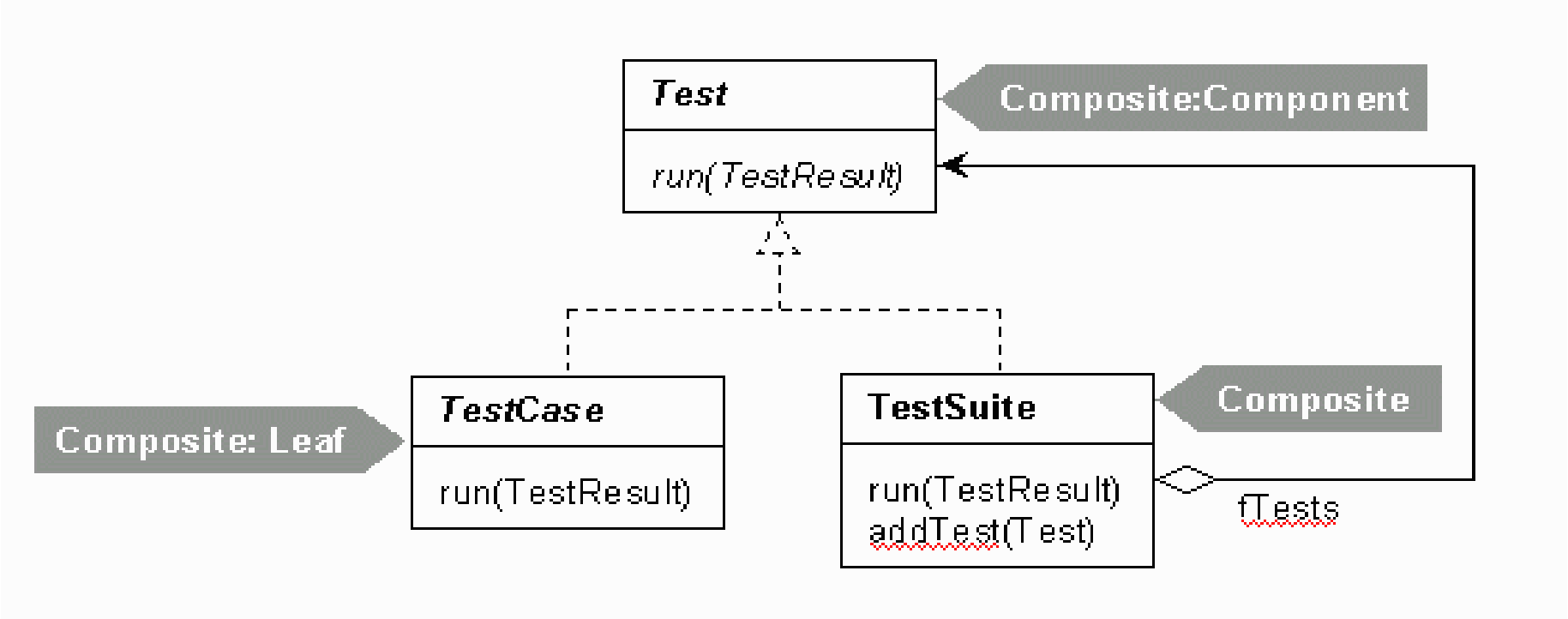
2. Einfache Elemente implementieren die Nutz-Schnittstelle direkt (z.B. toString)

3. Zusammengesetzte Elemente sind Behälter ihrer Bestandteile. Sie implementieren
   die Nutz-Schnittstelle im wesentlichen durch Delegation an ihre Bestandteile,
   die selbst wieder einfach oder zusammengesetzt sein können.



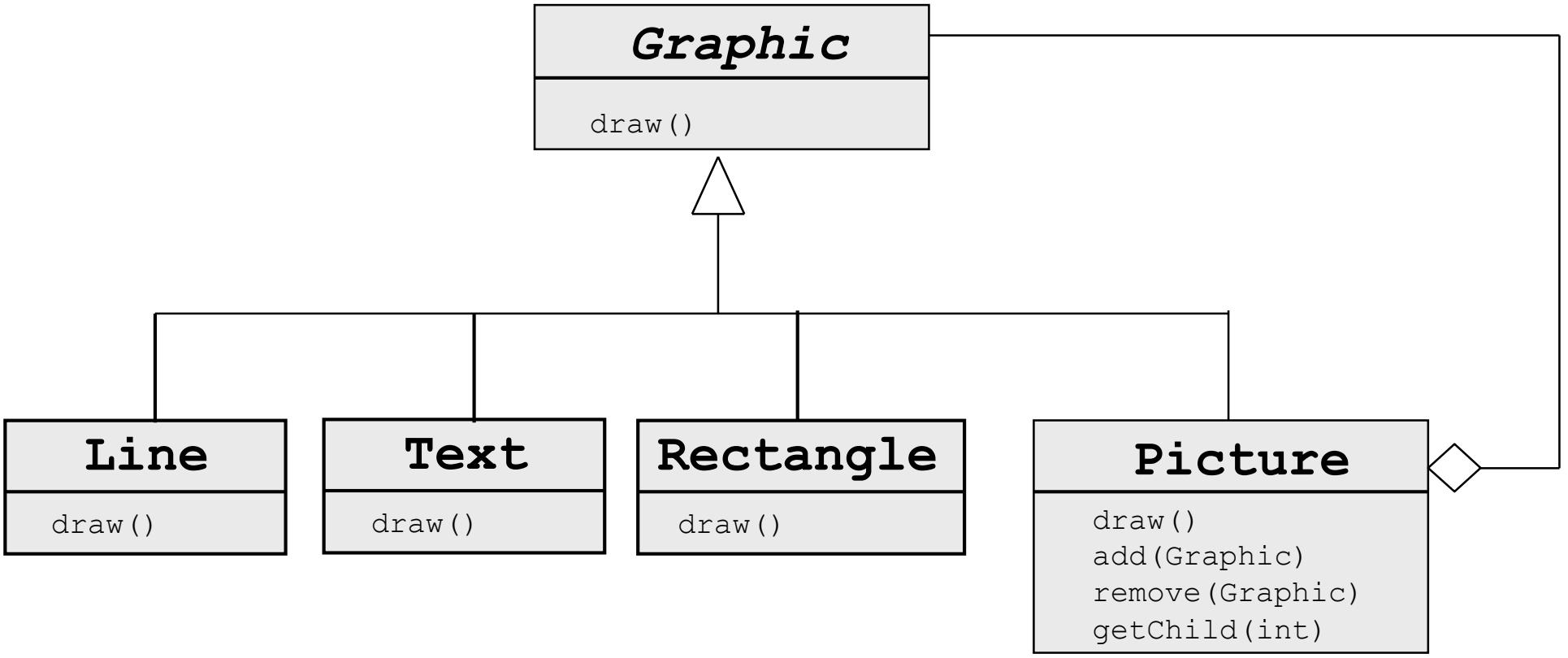

```
public class Container implements Component{
   Collection myElements;
   ..
   public String toString() {
     StringBuffer buf = new StringBuffer();
     Iterator i = myElements.iterator();
     while (i.hasNext())
         buf.append(i.next().toString());
     return buf.toString();
   }
   ..
}
```

Beispiel: Komposition in JUnit, Test und TestSuite



- *TestCase* implementiert *Test* wie beschrieben.
- *TestSuite* implementiert *Test* durch Iteration und Aufruf über die Menge der Tests (*fTests*);
  Tests werden über *addTest* hinzugefügt.

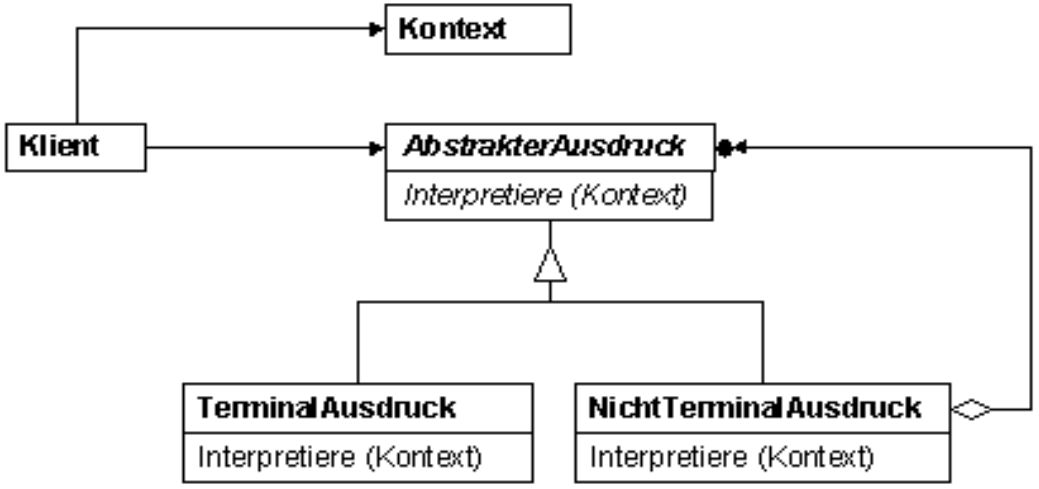
Weiteres Beispiel für Komposition



Komposition: Beziehung zu anderen Mustern

Interpreter-Muster

- Jeder Ausdruck ist terminal oder nicht-terminal (enthält andere Ausdrücke).
- Die Nutz-Schnittstelle enthält nur die Methode `interpret(context)`.
- Jeder terminale Ausdruck weiß selbst, wie er zu interpretieren ist;
- nicht-terminale Ausdrücke delegieren ihre Interpretation an ihre Bestandteile.

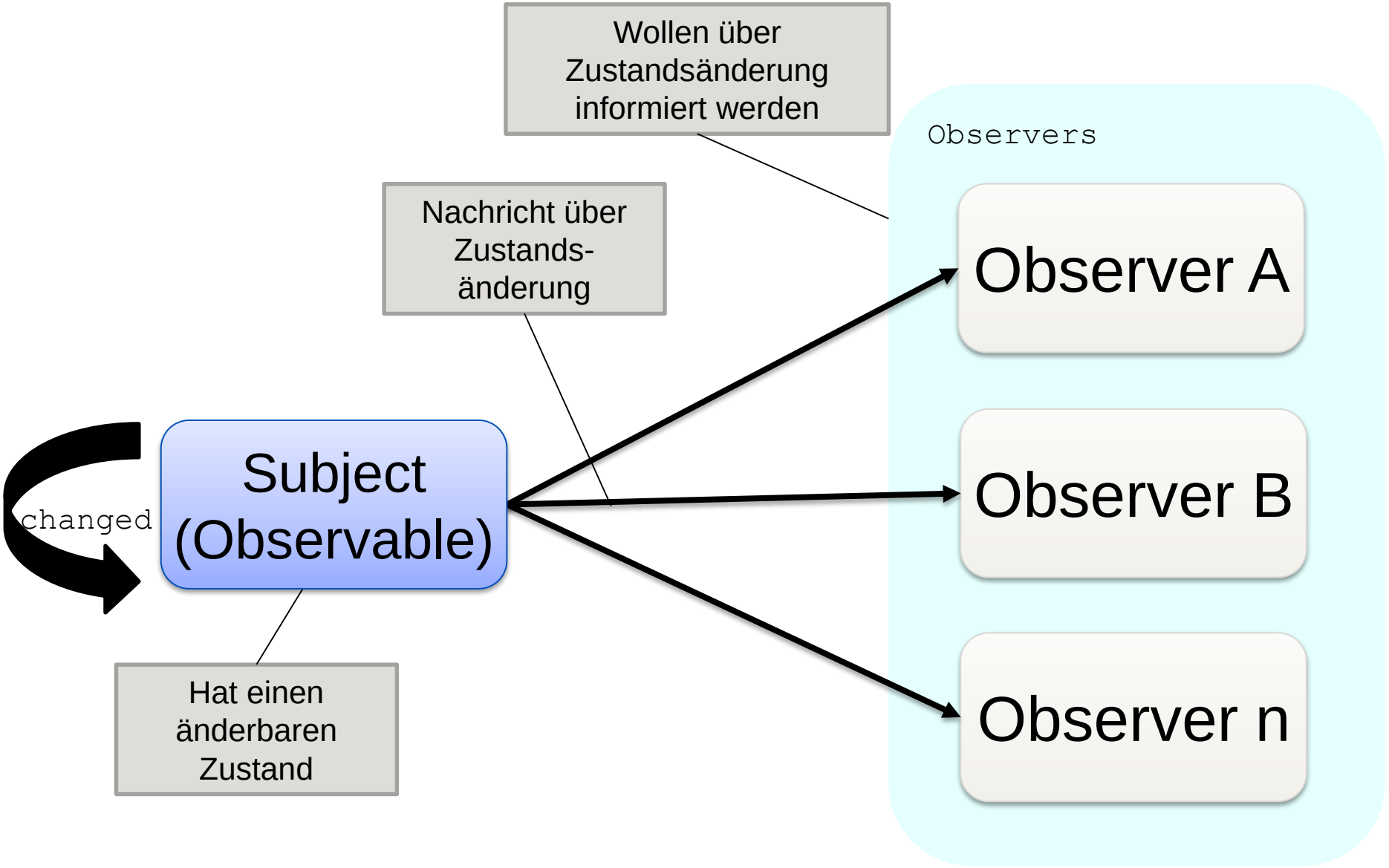


Observer-Muster = Herausgeber + Abonnenten

- Wenn Sie wissen wie das Abonnieren von Zeitungen vor sich geht, haben Sie im Grunde auch das Observer-Muster (Beobachter-Muster) verstanden.

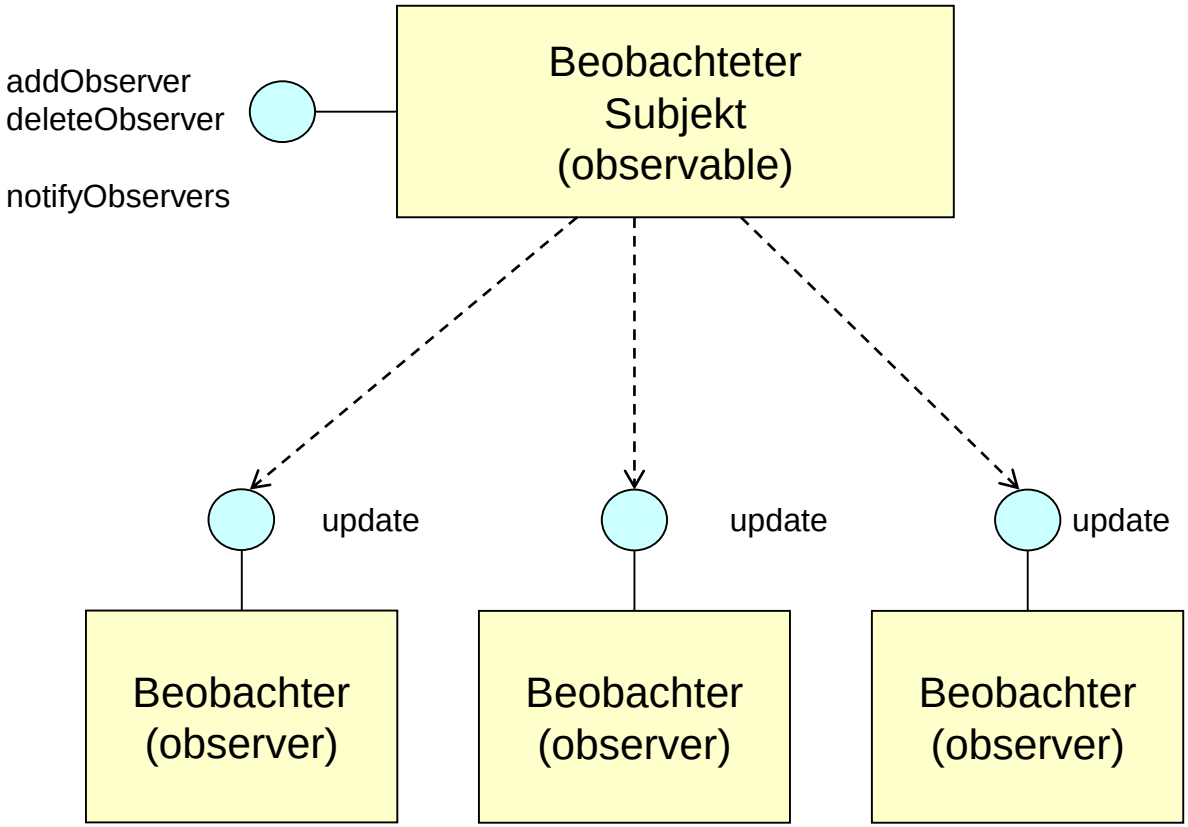
- So funktionieren Zeitungsabos
  - Verlag veröffentlicht eine Zeitung
  - Sie abonnieren eine Zeitung
    Wenn es eine neue Ausgabe gibt werden Sie automatisch beliefert
  - Sie kündigen das Abonnement
  - Solange der Herausgeber die Zeitung veröffentlicht abonnieren andere Personen oder Unternehmen die Zeitung oder kündigen das Abonnement

Observer-Muster: schematische Darstellung



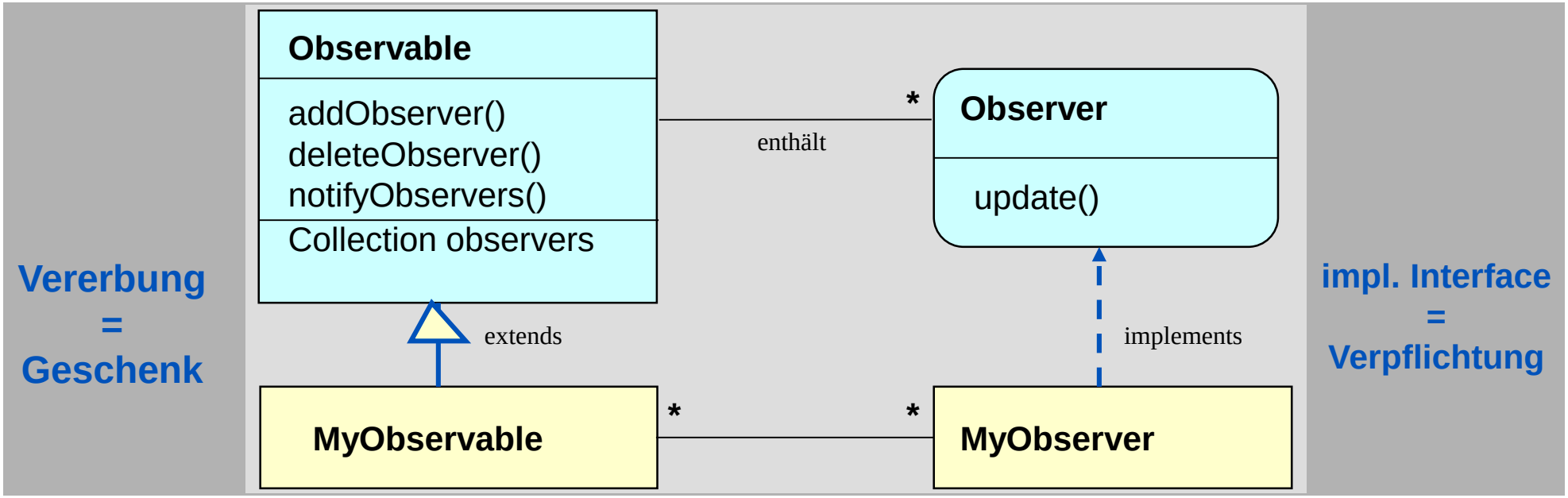
Beobachter-Muster: Schnittstellen

- der Beobachtete (Observable, Publisher, Subjekt) verwaltet Informationen
- die Beobachter (Observer, Subscriber) wollen über Änderungen informiert werden



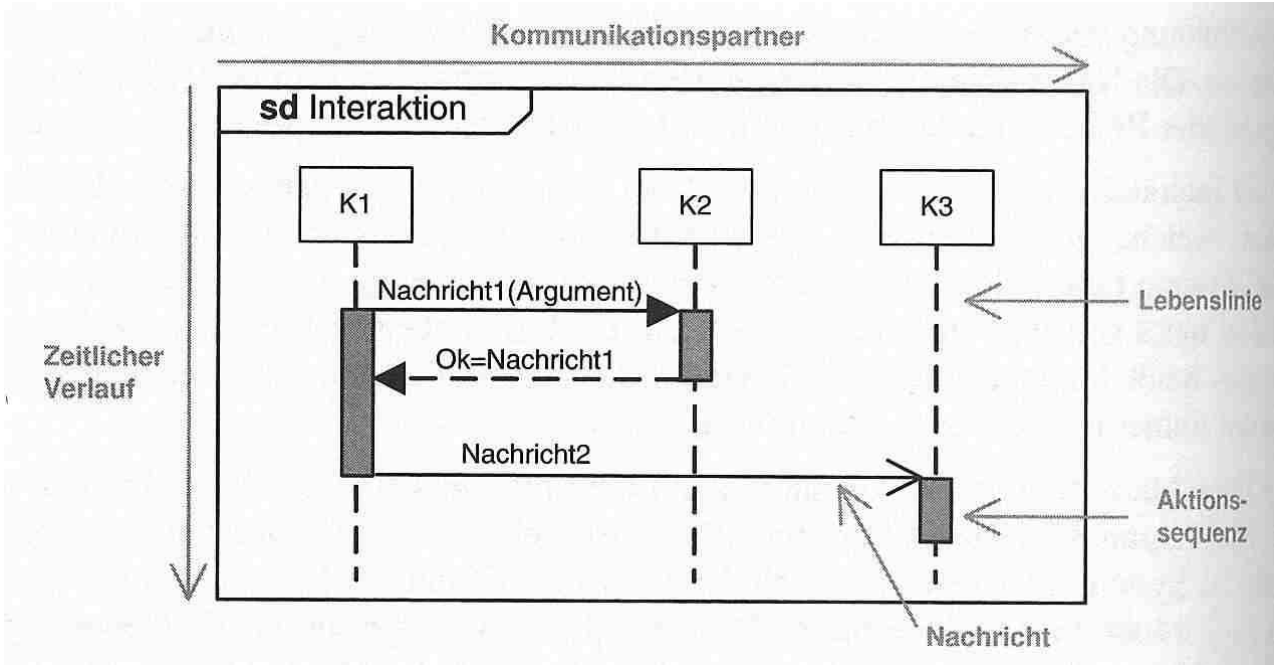
Beobachter-Muster: Klassendiagramm

- in Java: JDK enthält **`Interface Observer`**, enthält nur 1 Methode:
  `void update(Observable obv, Object info)`
- jede Observer-Klasse implementiert dieses Interface nebenberuflich
- der Observable verwaltet einen Array von Observern mittels **`addObserver`**`(Observer obs)` und **`deleteObserver`**`(Observer obs)`
- **`notifyObserver`**: Observable informiert Observers über `update`



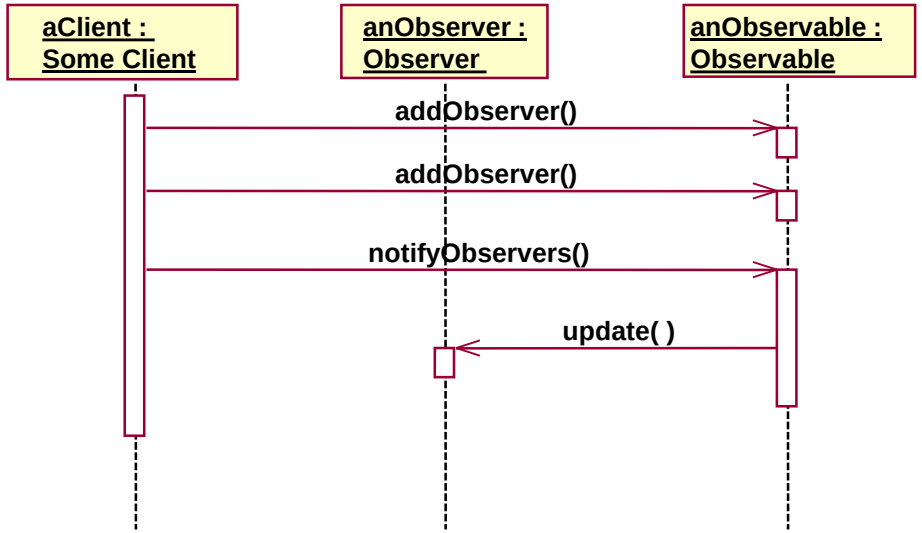
Kurzer Exkurs: Sequenzdiagramm in UML

- Elemente im Sequenzdiagramm
  - Kommunikationspartner als Lebenslinien
  - Nachrichten in Form von Pfeilen



[UML glasklar, dpunkt.verlag]

Beobachter-Muster: Sequenzdiagramm



Anwendung

- Wenn Änderungen eines Objektes Änderungen andere Objekte verlangen
- Benachrichtigung von Objekten ohne enge Kopplung
- Beispiele:
  - Zeitgeber aktualisiert Beobachter (Analog Uhr, Digital Uhr)
  - MVC (Model View Controller): Model ist Beobachteter, view ist Beobachter
  - Börsenticker

Beispiel Beobachter-Muster: 1. Observable

```
public class GetraenkeLager extends Observable {
  private String name;
  private int maxBier;
  private int anzahlBier;

  public GetraenkeLager(String name, int maxBier) {
    this.name = name;  this.maxBier = maxBier;
    auffuellen();
  }

  public int getAnzahlBier()    { return anzahlBier; }
  public String toString() {
    return "Getränklager: "+name+" enthält "+anzahlBier+" Bier.";
  }

  public void auffuellen() {
    anzahlBier = maxBier;
  }

  public void entnehme(int biere) {
    if (anzahlBier >= biere) {
      System.out.println("\n"+biere+" Bier aus "+name+ " entnommen.");
      anzahlBier -= biere;
      setChanged( );
      notifyObservers( );
    }
  }
}
```

Beispiel Beobachter-Muster: 2. Observer

```
public class LagerVerwalter implements Observer {

  private int minBiere;
  private String name;


  public LagerVerwalter(String name, int minBiere) {
    this.name = name;
    this.minBiere = minBiere;
  }


  public void update(Observable o, Object arg) {
    GetraenkeLager lager = (GetraenkeLager) o;
    System.out.println(name+": schau mer mal");
    System.out.println("    "+lager.toString());
    if(lager.getAnzahlBier() <= minBiere) {
      System.out.println("    "+ name + ":oha, des langt aber nimmer,"
                         + " gleich wieder auffüllen");
      lager.auffuellen();
      System.out.println("    "+lager.toString());
    } else {
      System.out.println("    "+name+":ois ok");
    }
  }
}
```

Beispiel Beobachter-Muster: 3. Anwendung

```
public class TestParty extends TestCase {

  public void testParty() {
    // party vorbereiten
    GetraenkeLager kuehlschrank = new GetraenkeLager("Kühlschrank", 6);
    GetraenkeLager balkon = new GetraenkeLager("Balkon", 100);
    GetraenkeLager badewanne = new GetraenkeLager("Badewanne", 20);

    LagerVerwalter gastgeber = new LagerVerwalter("Gastgeber", 3);

    kuehlschrank.addObserver(gastgeber);
    balkon.addObserver(gastgeber);
    badewanne.addObserver(gastgeber);

    // viele Gäste kommen und trinken: max, karl, josef, gabi, susi
    kuehlschrank.entnehme(5); // max holt für alle am Tisch Getränke

    // guter Kumpel kommt und passt mit auf Kühlschrank und Badewanne auf
    LagerVerwalter guterKumpel = new LagerVerwalter("GuterKumpel", 0);
    kuehlschrank.addObserver(guterKumpel);
    badewanne.addObserver(guterKumpel);

    kuehlschrank.entnehme(3); // max holt die nächsten drei Bier

    badewanne.entnehme(1); // karl holt sich ein Bier aus der Badewanne
  }
}
```

Observer-Muster: Anmerkungen

- Damit ein Objekt ein Beobachter wird
  - Die Schnittstelle **`Observer`** implementieren
- Damit ein Subjekt Benachrichtigungen sendet
  - Die Superklasse **`Observable`** erweitern
  - **`setChanged( )`** aufrufen
  - **`notifyObservers( )`** oder **`notifyObservers(Object args)`** aufrufen
- Observer-Muster bietet ein Objektdesign bei dem Subjekt und Beobachter locker gebunden sind
  - Beide können interagieren, müssen aber wenig Kenntnisse voneinander besitzen

Singleton-Muster

- In vielen Anwendungsfällen will man nur eine Instanz einer Klasse erlauben
- Wichtige Eigenschaften:
  - Kein `public` Konstruktor
  - Erzeugung über statische Methoden

```
class Singleton
{
  private static final Singleton instance =
                          new Singleton( );

  private Singleton( )
  {
    //...
  }

  public static Singleton getInstance()
  {
    return instance;
  }
}
```

- Anwendungen: Verwalter-Klassen, Subsystemfassaden, System-Resourcen, ... (z.B. Drucker-Spooler, GUI-Window Manager, File System Manager), gemeinsame Warteschlange, MessageQueue

Singleton-Muster: Anmerkungen

- Singleton sichert, dass eine Anwendung nur eine Instanz einer Klasse hat
- Singleton bietet einen globalen Zugriffspunkt auf diese Instanz
- Vorteil vom Singleton:
  - Damit können zentral Ressourcen oder Objekte verwaltet werden, da sichergestellt ist, dass es nur ein Verwaltungsobjekt gibt.
  - Überall im Code kann mit getInstance( ) auf das Singleton zugegriffen werden, ohne das Objekte erzeugt oder zerstört werden müssen.
- Achtung: Singleton ist ein sehr beliebtes (Alibi) Muster da es so leicht zu implementieren ist.
  → behutsam einsetzen
- Vorsicht bei Multithreading! Alle Threads verwenden dann die gleiche Instanz.

Fliegengewicht-Muster: Idee

- Für manche Problemstellungen ist es elegant, auch sehr kleine „Entitäten“ als Objekte zu modellieren, dies kann jedoch zu einer Unmenge von Objekten führen, was die Effizienz stark negativ beeinflusst.

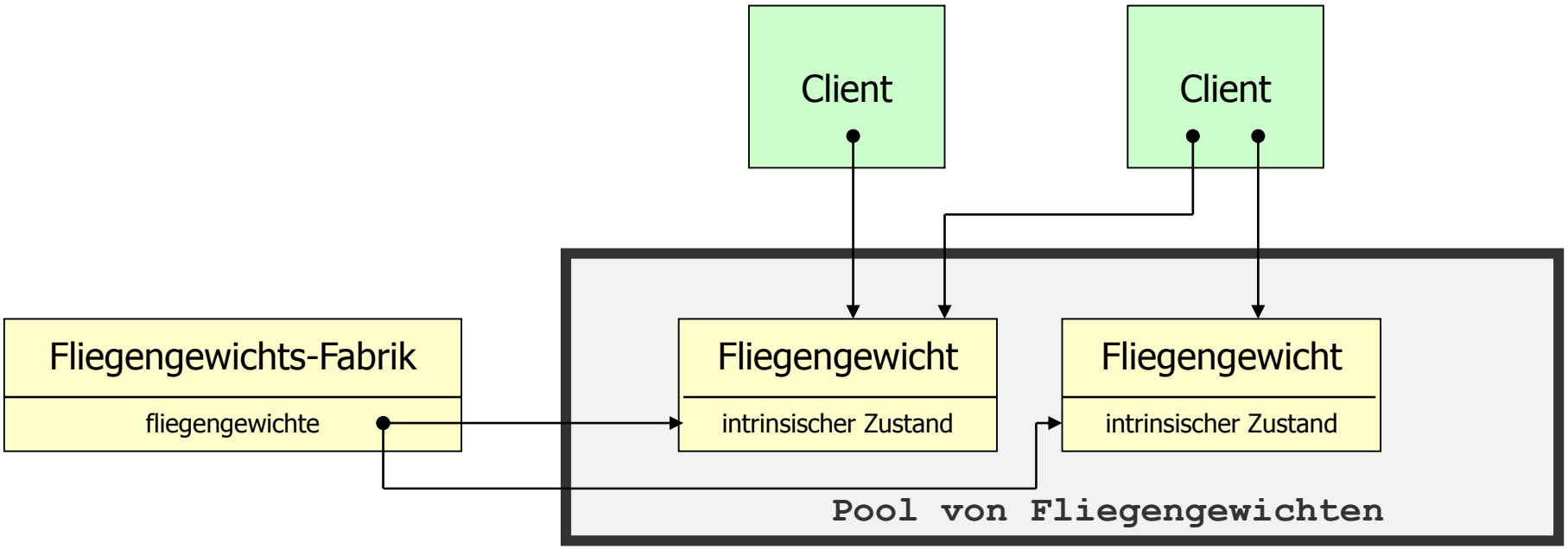
  Ein **Fliegengewicht** (flyweight) ist ein (leichtgewichtiges) Objekt, das in verschiedenen Kontexten effizient verwendet werden kann.

- **Idee**
  - Ein Fliegengewicht wird in mehreren Kontexten gleichzeitig verwendet, jedoch in einer Art, dass es für den Verwender von „normalen“ Objekten nicht unterscheidbar ist.
  - Dadurch wird die Anzahl nötiger Objekte drastisch reduziert.
  - Fliegengewichte dürfen daher keine Annahmen über den Kontext machen, in dem sie verwendet werden.

Fliegengewicht-Muster: Umsetzung

- Implementierung von Fliegengewichten
  - Aufteilung des Zustands der Objekte in
  - **Intrinsischen Zustand**: ist vom Verwendungskontext unabhängig und wird im Fliegengewichtsobjekt abgespeichert.
  - **Extrinsischer Zustand**: Hängt vom Verwendungskontext ab und kann daher nicht mehrfach verwendet werden und somit nicht im Fliegengewicht abgespeichert werden. Verwender des Fliegengewichts muss diesen Zustand dem Fliegengewicht bei jeden Methodenaufruf übergeben. Extrinsischer Zustand kann häufig entweder gespeichert oder berechnet werden (oder eine Mischung).
- Bemerkungen
  - Fliegengewichte werden i.allg. durch Fabriken erzeugt (→ Wiederverwendung)
  - Fliegengewichte sind häufig (aber nicht immer) unveränderlich (immutable)

Fliegengewicht-Muster: Klassendiagramm



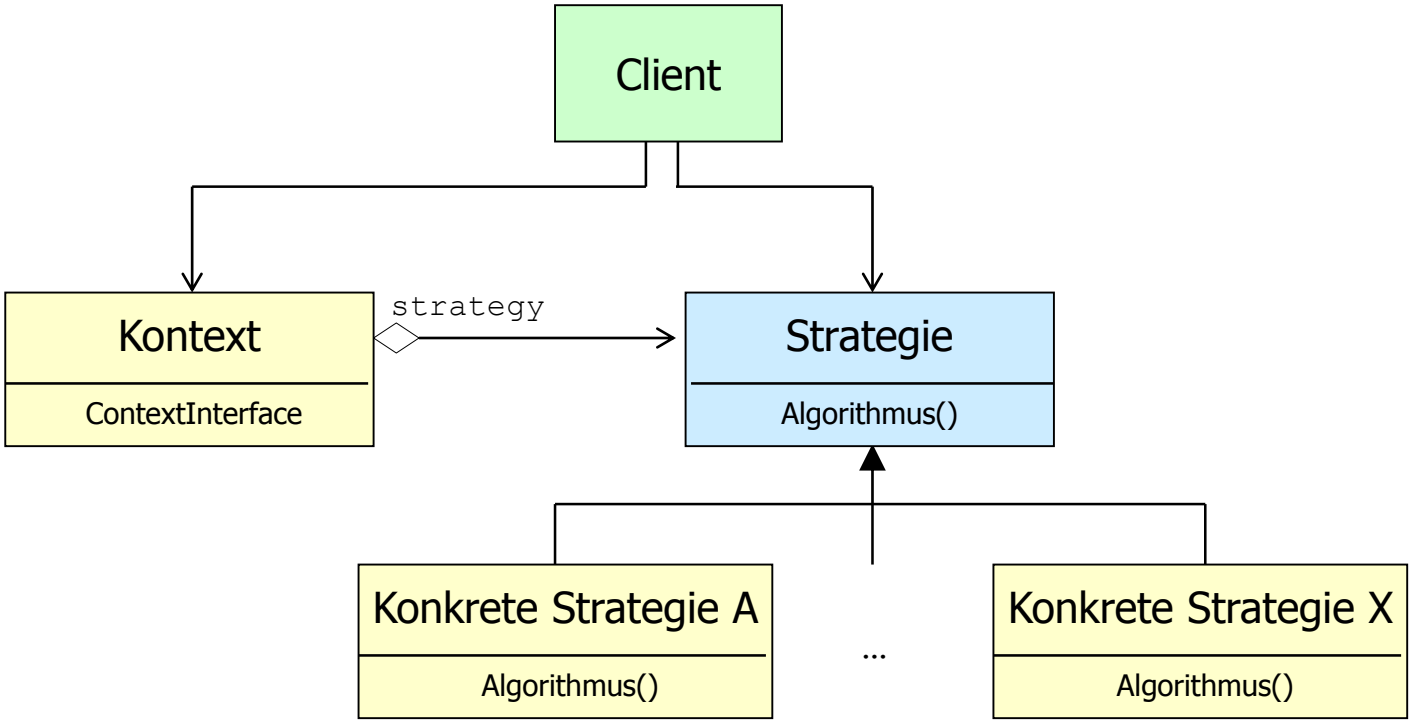
- Wichtige Punkte
  - Kern ist das Erkennen von intrinsischem und extrinsischem Zustand

Strategie-Muster: Idee

- Das **Strategie-Muster** definiert eine Klasse von austauschbaren Algorithmen. Implementierung i.allg. über eine Mengen von Klassen, die ein bestimmtes Interface implementieren.

- Sinnvolle Anwendungsfälle
  - viele verwandte Klassen unterscheiden sich nur in ihrem Verhalten.
  - unterschiedliche (austauschbare) Varianten eines Algorithmus werden benötigt.
  - Daten innerhalb eines Algorithmus sollen vor Nutzern verborgen werden.
  - Komplexe Datenstrukturen liegen bei den Algorithmen und nicht beim Client
  - verschiedene Verhaltensweisen sind innerhalb einer Klasse fest integriert (meist über Mehrfachverzweigungen).

Strategie-Muster: Klassendiagramm



- Kontext
  - wird mit der Konkreten Strategie konfiguriert (hat also ein Attribut vom Typ „Strategie“)
  - hat ein Interface, über das die Strategie nötige Daten anfragen kann.

Strategie-Muster: Code Beispiel für Enten mit Flugstrategie

```
public abstract class Ente {
    FlugVerhalten fv;

    public void tuFliegen(){
        fv.fliegen();
    }
    public void setFlugVerhalten(FlugVerhalten fv){
        this.fv = fv;
    }
    . . .
}
```

```
public interface FlugVerhalten{
    public void fliegen( );
}
```

```
public class FliegtMitFluegel implements FlugVerhalten {
    public void fliegen(){
        System.out.println(„Ich fliege“);
    }
}
```

```
public class FliegtGarNicht implements FlugVerhalten {
    public void fliegen(){
        System.out.println(„Ich kann nicht fliegen“);
    }
}
```

Setzen der Strategie und Fliegen

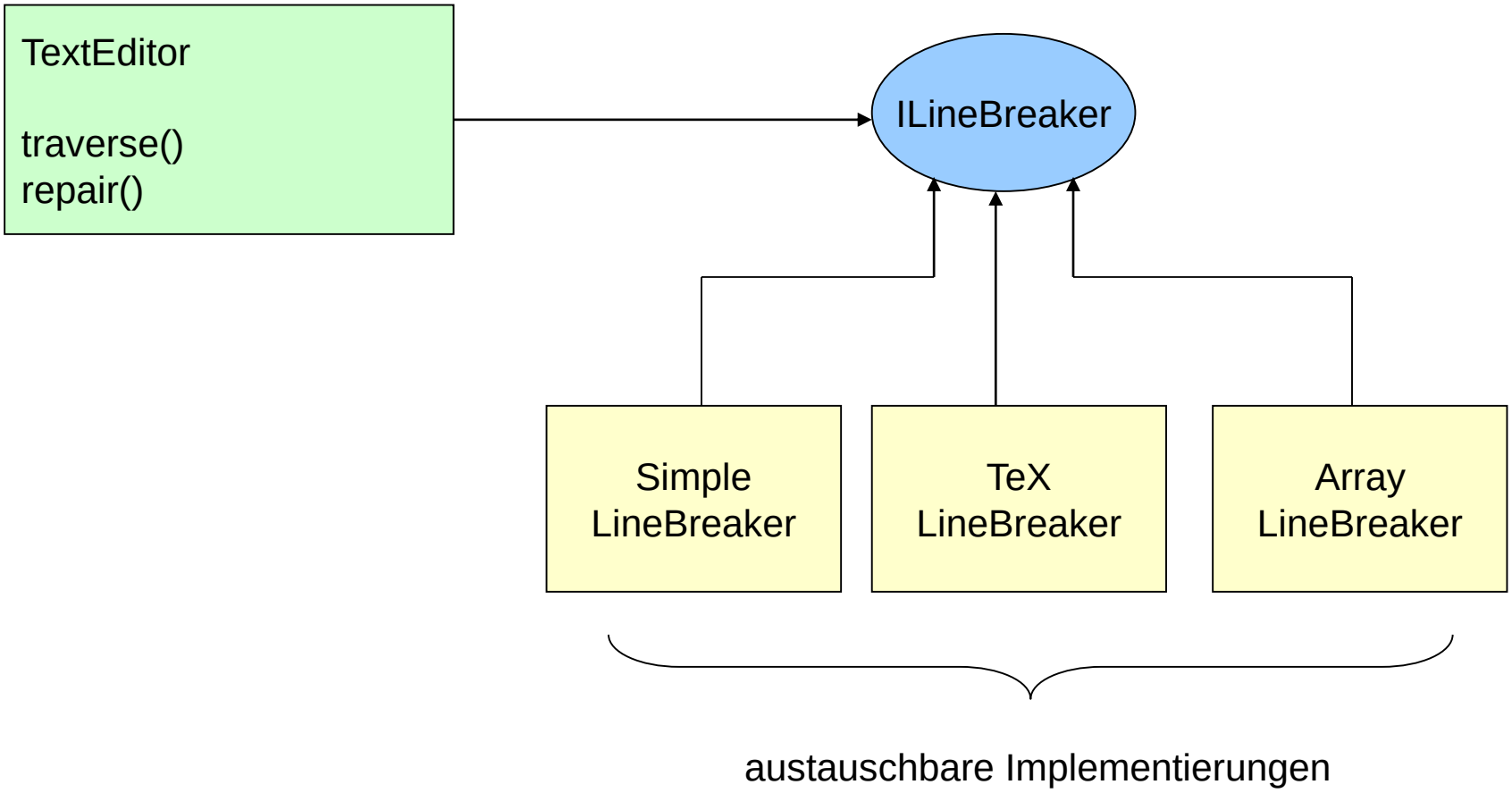
```
public class StockEnte extends Ente{
   public StockEnte(){
       fv = new FliegtMitFluegel();
   }
   . . .
}
```

```
public class HolzEnte extends Ente{
   public HolzEnte(){
       fv = new FliegtGarNicht();
   }
   . . .
}
```

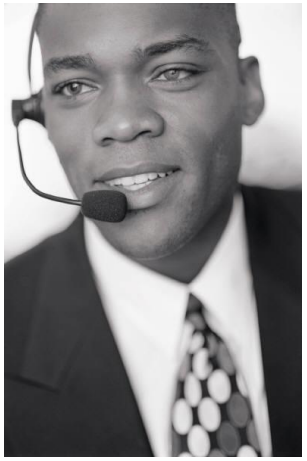
```
public class EntenSimulator {
    public static void main(String args) {
        Ente donald = new StockEnte();
        donald.tuFliegen();
        donald.setFlugVerhalten(new FliegtGarNicht());
        donald.tuFliegen();
        Ente dagobert = new HolzEnte();
        dagobert.tuFliegen();
}
```

Strategie = Interface + austauschbare Implementierung

Beispiel: Texteditor



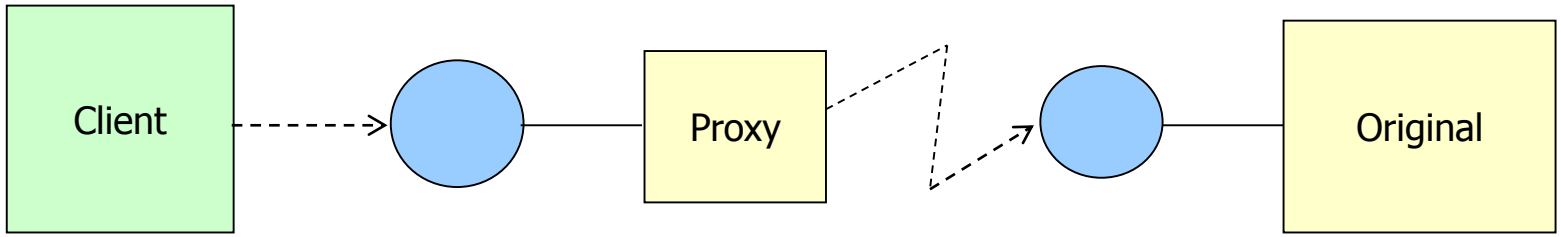
Proxy-Muster: Idee

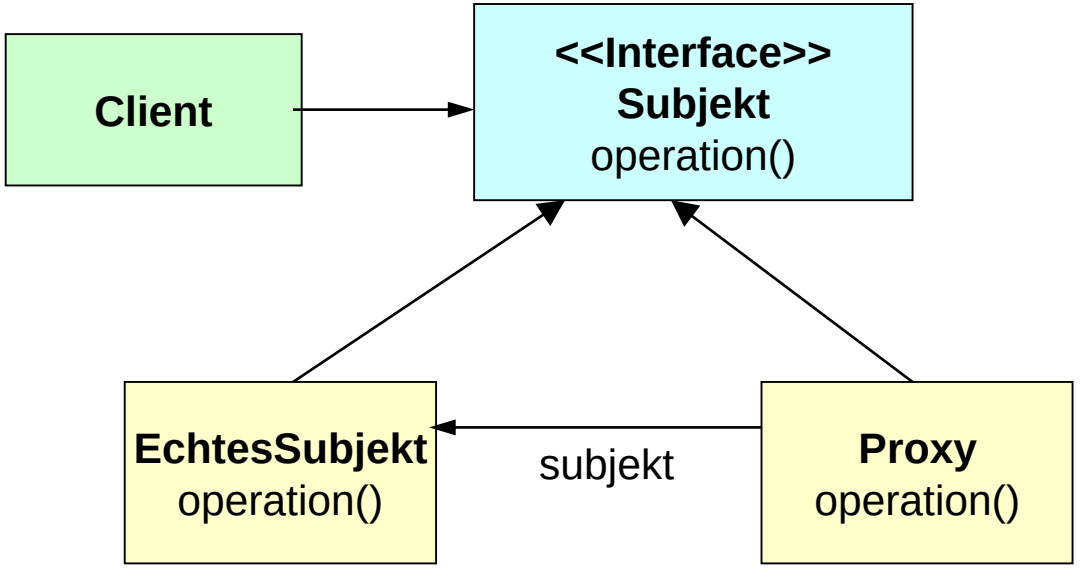


- Proxies sind lokale Platzhalter (oder Repräsentanten, Stellvertreter) von Objekten des Originals, auf die man nicht ohne weiteres zugreifen kann, weil sie sich z.B. auf der Datenbank oder in einem anderen Prozeß befinden.

- **Idee**:
  - Der Importeur sieht nur die Schnittstelle des Originals.
  - Der Proxy implementiert diese Schnittstelle, indem er die Anfragen an das Original weiterleitet.
  - Der Proxy kontrolliert den Zugriff auf das Originalobjekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.
  - Für den Importeur ist es bis auf Initialisierung, Performance und Robustheit transparent, ob er das Original aufruft oder den Proxy.

Proxy-Muster: Klassendiagramm





Proxy-Muster: Code Beispiel für einen Cache-Proxy

```
public class OnlineServiceCenter implements IStudentenverwaltung{
    . . .
    public Student getStudent(String matrikelnummer){
          // Aufwändiger Zugriff über OSS
    }
    . . .
}
```

```
public interface IStudentenverwaltung{
    public Student getStudent(String matrikelnummer);
}
```

```
public class StudentenCacheProxy implements IStudentenverwaltung{
    private IStudentenverwaltung oss;
    private StudentenCache cache;

    public StudentenCacheProxy(IStudentenverwaltung oss, StudentenCache cache) {
        this.oss = oss;
        this.cache = cache;
    }
    public Student getStudent(String matrikelnummer){
        Student s = cache.getStudent(matrikelnummer);
        if (s == null){
            s = oss.getStudent(matrikelnummer);
            cache.addStudent(s);
        }
    }
}
```

Proxy-Muster: Bewertung

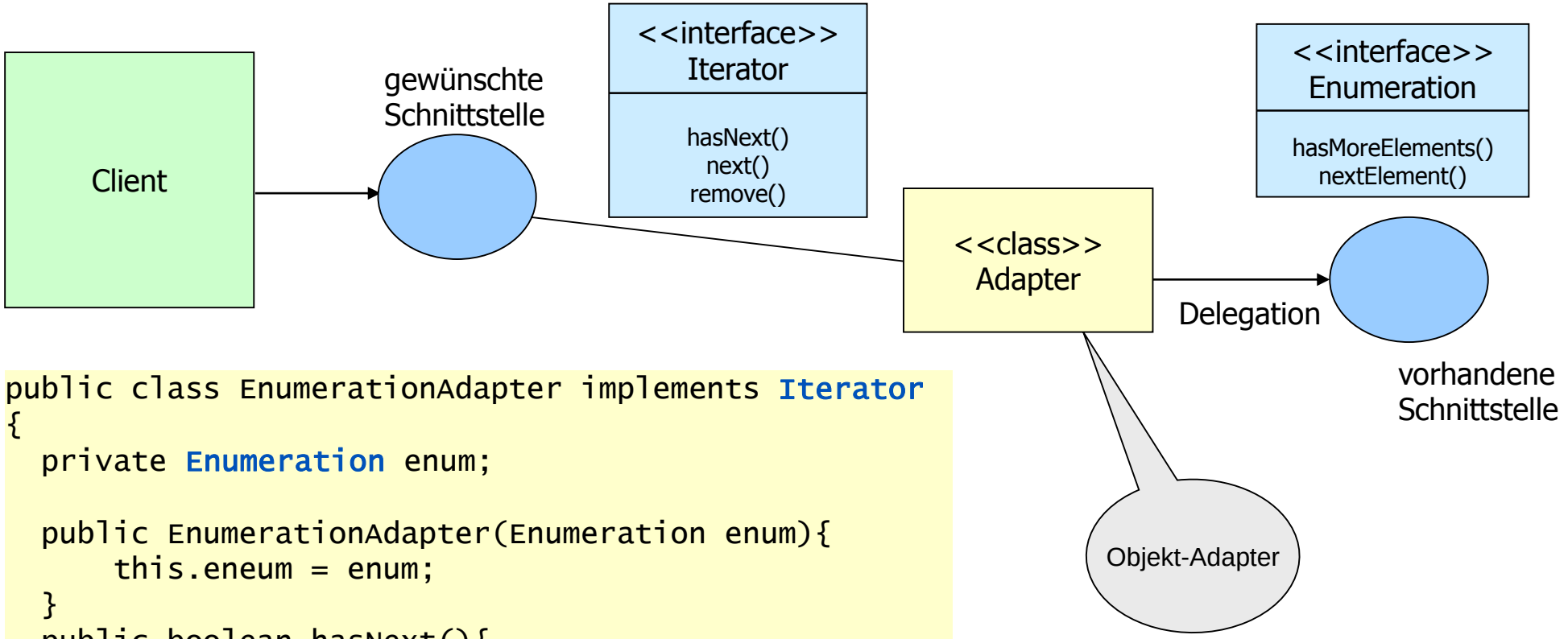
- Proxies sind anpassungsfähiger und intelligenter als Zeiger

- **Anwendungen**
  - Remote-Proxy: lokaler Stellvertreter für ein Objekt, remote-Zugriff nur bei Bedarf
  - Virtueller Proxy: erzeugt teure Objekte nur bei Verlangen
  - Schutz Proxy: Kontrolliert Zugriff auf Originalobjekt
  - Smart Reference (z.B. Zählen von Referenzen auf Objekt)
  - Cache-Proxy: Ermöglicht vorübergehende Speicherung der Ergebnisse von aufwändigen Operationen in einem Puffer
  - Sicherheitsfassade (s. Kapitel Fehler und Ausnahmen): Übernimmt die Fehlerbehandlung einer Komponente

Adapter-Muster: Idee

- Adapter (Wrapper) konvertieren die Schnittstelle einer Komponente in eine andere Schnittstelle, die den Erwartungen der aufrufenden Komponente entspricht.
- Adapter werden häufig zur Einbindung von Fremdsystemen eingesetzt
- Voraussetzung: Die vorhandene Schnittstelle passt wenigstens ungefähr.
- **Idee:**
  - Der Importeur sieht nur die Schnittstelle des Adapters.
  - Der Adapter implementiert diese Schnittstelle, indem er Anfragen konvertiert und an die vorhandene Komponente weiterleitet.
  - Der Adapter kann entweder von der vorhandenen Komponente erben (Klassenadapter) oder er kann wie ein Proxy an die vorhandene Komponente delegieren (Objektadapter).

Adapter-Muster: Klassendiagramm und Codebeispiel




```
public class EnumerationAdapter implements Iterator
{
  private Enumeration enum;

  public EnumerationAdapter(Enumeration enum){
      this.eneum = enum;
  }
  public boolean hasNext(){
     return enum.hasMoreElements();
  }
  public Object next(){
     return enum.nextElement();
  }
  public void remove(){
     throw new UnsupportedOperationException();
  }
}
```

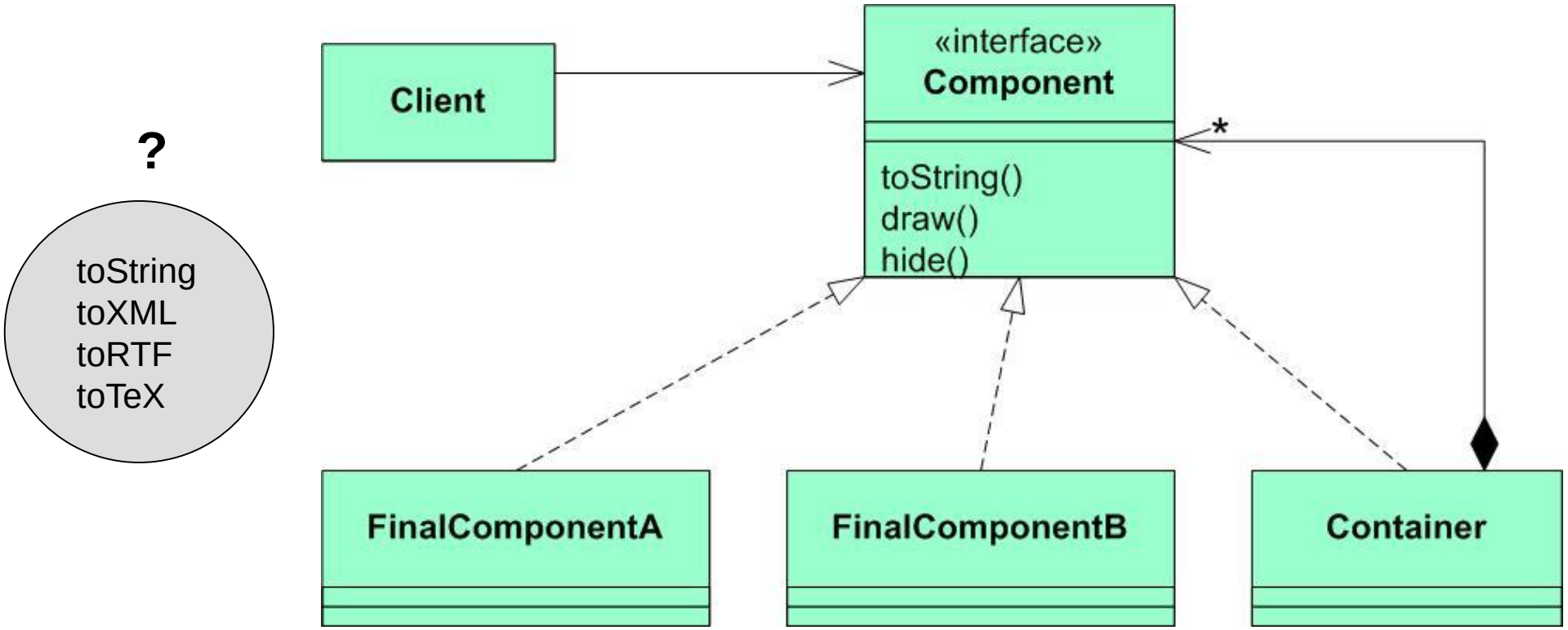
Adapter-Muster: Bewertung

Vorteile:

- Unabhängigkeit von der konkreten Implementierung
- Vereinfachung: Aus einer maximalen Schnittstelle wird die genau passende
- Ermöglicht Wiederverwendung und Zusammenarbeit mit Klassen die keine passende Schnittstelle besitzen

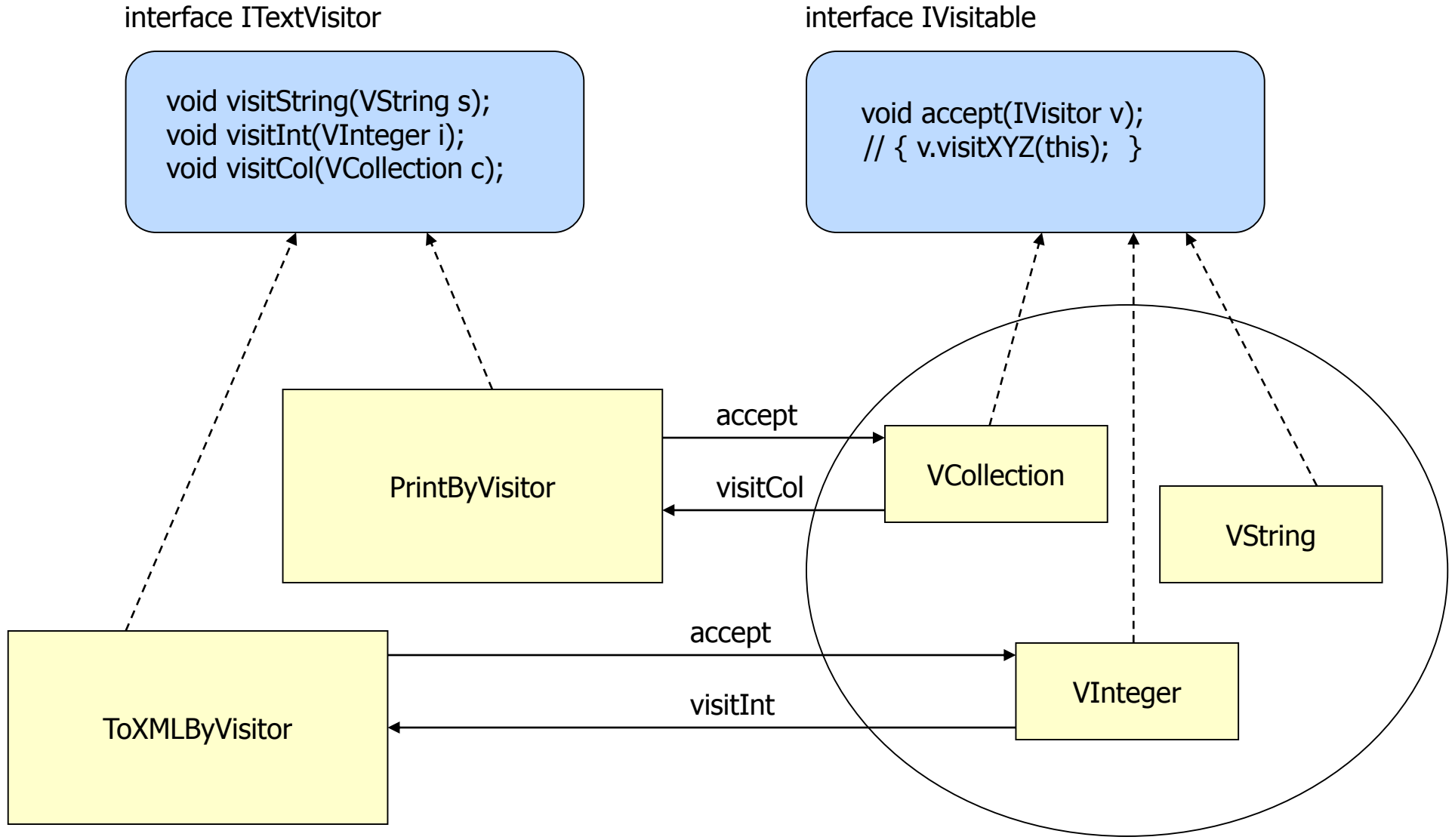
Besucher-Muster (Visitor)

Problem: **Mit den Klassen einer Komposition verschiedene Dinge tun!**



Naiver Ansatz: Alle Klassen der Komposition kennen alle Darstellungen!?

toXML by Visitor



Visitor-Muster: Code-Beispiel

Component

```
public class VInteger implements
                      ITextElement, IVisitable{
    . . .
    void accept(ITextVisitor visitor){
       visitor.visitInt(this);
    }
    . . .
}
```

```
ToXMLByVisitor visitor =
           new ToXMLByVisitor();
List<ITextElement> text = . . .;
for (ITextElement el : text)) {
  el.accept(visitor);
}
String xml = visitor.getXML;
```

Client

```
public class ToXMLByVisitor extends ITextVisitor {
   private String xml;

   public String getXML() {
      return xml;
   }
   public void visitInt(VInteger i) {
     xml += „<Integer>“ + i.toString() + „</Integer>“;
   }
   public void visitCol(Vcol col) {
     for(IVisitable el : col.getElements())
         el.accept(this);
   }
   . . .
}
```

Visitor

Besucher-Muster: Bewertung

Anwendung

- Objektstruktur mit vielen Klassen und unterschiedlichen Schnittstellen
- Viele unterschiedliche nicht miteinander verwandte Operationen auf einer Objektstruktur
- Klassen der Objektstruktur ändern sich selten, aber häufig neue Operationen
  → Hinzufügen neuer Operationen einfach
  → Hinzufügen neuer Klasse aufwendig
- Beispiel: Syntaxbaum mit verschiedenen Operationen darauf
- Achtung:
  - Häufig hat Besucher eine schlechte Performance (Verdopplung der Anzahl der Aufrufe)
  - Die Kapselung der Kompositumklasse wird zerstört

Zusammenfassung Muster (1)

- Jeder fortgeschrittene Programmierer muss die wichtigsten Muster verstehen und anwenden können
- Die richtige Verwendung von Mustern macht große Programmsysteme lesbarer
- Muster wenden die wesentlichen Prinzipien von guter objektorientierter Programmierung an
- Außer den behandelten existieren noch viele weitere Muster
  - Erzeugen von Objekten, z.B. Abstrakte Fabriken, Prototypen, Builder, …
  - Strukturelle Muster, z.B. Bridge, Decorator, Facade, …
  - Verhaltensmuster, z.B. Interpreter, Kommando, …
  - Muster für spezielle Anwendungen, z.B. für Enterprise Applikations, für SQL,
  - Muster für GUI, z.B. Model-View-Controller, Model-View-Presenter

Zusammenfassung Muster (2)

- Hier behandelte Muster
  - **Iterator-Muster**: Iteration über Elemente
  - **Fabrik-Muster**: Erzeugen von konkreten Objekten eines Interfaces, ohne die konkrekten Klassen zu kennen
  - **Composite-Muster**: gut geeignet zur Bearbeitung rekursiver Objektstrukturen
  - **Observer-Muster**: wird häufig eingesetzt (z.B. bei graphischen Oberflächen)
  - **Singleton-Muster**: sehr einfach zu implementieren, es sollte aber behutsam eingesetzt werden
  - **Fliegengewicht-Muster**: Objekte in unterschiedlichen Kontexten verwenden, dadurch Objektanzahl drastisch reduzieren
  - **Strategie-Muster**: für austauschbare Algorithmen
  - **Proxy-Muster**: lokale Platzhalter von Original-Objekten, auf die man nicht ohne weiteres zugreifen kann
  - **Adapter-Muster**: Konvertieren von Schnittstellen
  - **Visitor-Muster**: mit den Klassen einer Komposition verschiedene Dinge tun