UML und Design-Patterns

Agenda

- Einführung: Was ist die UML? Was sind Entwurfsmuster?
- UML-Klassendiagramme

- OO-Techniken und Designpatterns
  - Abstrakte Klassen, Interfaces und deren Kombination

  - Template Method
  - Iterator
  - Adapter
  - Null-Object
  - Composite
  - Decorator
  - Strategy

Einführung

Was ist die UML?

- **UML** = **U**nified **M**odelling **L**anguage
- grafische Notationsform für die objekt-orientierte Analyse und Design (OOA/OOD)
- einheitliche Sprache bei der Architektur und der Modellierung von Software
- Begründer waren die „drei Amigos“
  Grady Booch, James Rumbaugh, Ivar Jacobson

Objekt-Guru	**Modellierungstechnik**
Grady Booch	OOAD - Object Oriented Analysis and Design
James Rumbaugh	OMT - Object Modeling Technique
Ivar Jacobson	OOSE - Object-Oriented Software Engineering

Was ist die UML?

- **Ziele der UML**
  - keine Festlegung auf spezielle Programmiersprache
  - universelle ***Beschreibungssprache*** für alle möglichen Arten von objektorientierten Softwaresystemen
  - standardisierte Sprache mit der Entwickler ihre ***Modelle beschreiben*** und mit anderen Entwicklern ***austauschen*** können
  - Alle Phasen der Softwareentwicklung abdecken: Anforderungsanalyse bis Timing-Verhalten
  - Beschreibung ***statischen*** und ***dynamischen*** Verhaltens

Strukturdiagramme

Verhaltensdiagramme

Was sind Entwurfsmuster?

- Definition Entwurfsmuster (Designpattern) =
  Bereits erprobte und mehrfach eingesetzte allgemeingültige Verfahren zur Lösung eines Entwurfsproblems

- Begründer ist die sogenannte Gang Of Four (GOF)
  Gamma, Helm, Johnson, Vlissides

- Populär durch das GOF-Buch ,,Designpatterns - Elements of reusable Software"

- Entwurfsmuster beschreiben prototypischen Beispielumsetzungen
  - Problem auf eine dokumentierte Art und Weise lösen
  - Entwurfsmuster definieren eine eigene Entwurfssprache

UML und Entwurfsmuster

Das kongeniale Duo: UML und Entwurfsmuster

- Früher:

  Wollte man einem anderen Software-Entwickler eine Realisierung beschreiben, Dazu waren **viele Worte und Zeichnungen nötig + unterschiedliche Notationen und kaum gemeinsame Entwurfssprache!**

- Heute:
  - UML bietet uns, eine standardisierte Notation
  - Entwurfsmuster helfen uns, Designentscheidungen zu kommunizieren

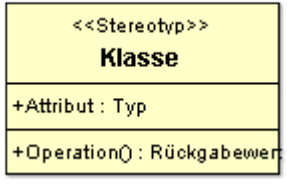
UML-Klassendiagramme

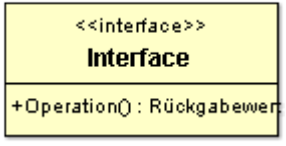
- **Klassendiagramme**
  - **Klassen, Interfaces, Abstrakte Klassen**
  - **Beziehungen zwischen Klassen**
    - Vererbung, Implementierung
    - Assoziationen
    - Aggregationen

Klassendiagramm

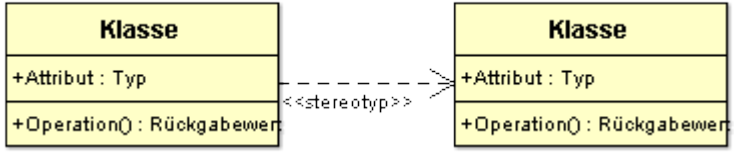
- **Ziel**: Beschreibt statische Elemente des Systems sowie deren Beziehung.

- **Wichtigste Bestandteile**:
  - **Klasse**: repräsentiert ein Element mit seinen Merkmalen (Attribute und Operationen). Die Art eines Elementes kann durch sogenannte Stereotypen näher spezifiziert werden.
  - **Interface**: repräsentiert eine Menge von (kohärenten) Operationen. Unterschied zur Klasse: das „was“ wird spezifiziert, nicht das „wie“. Entspricht konzeptionell einem „Typ“.



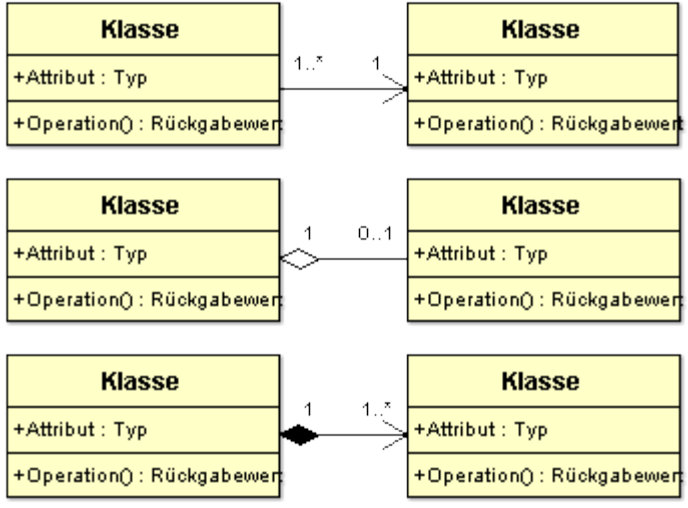


  - **Abhängigkeiten**: allgemein, die über Stereotypen konkretisiert werden kann.



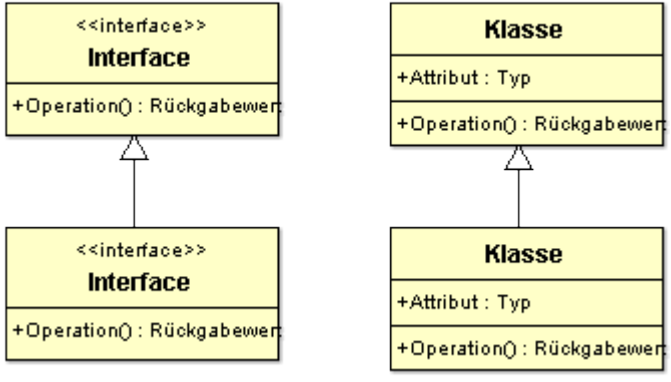
Klassendiagramm

- **Beziehungen zwischen Instanzen von Klassen**
  - **Assoziation**: Allgemeine Beziehung
  - **Aggregation**: Beziehung zwischen einem „Ganzen“ und seinen „Teilen“
  - **Komposition**: Beziehung zwischen einem „Ganzen“ und seinen „Teilen“, wobei die Lebenszeit der Teile an die des Ganzen gebunden ist.



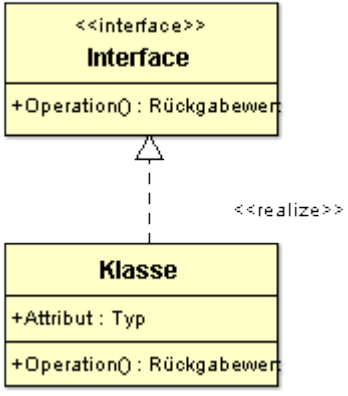
Klassendiagramm

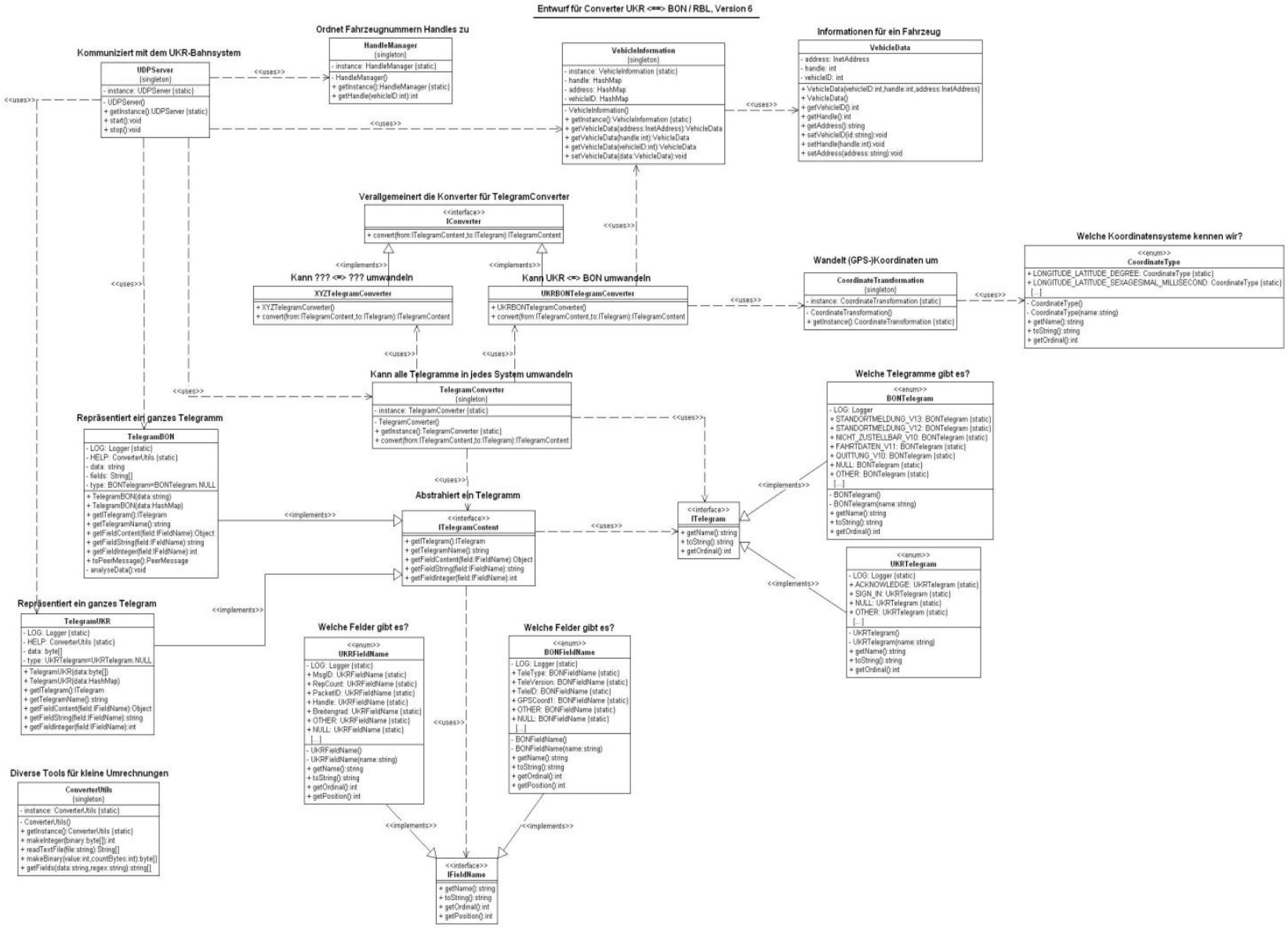
- **Generalisierung**:
  - Zwischen Klassen: Konzeptionell, eine Klasse ist eine spezielle Ausprägung einer anderen Klasse. Aus Nutzungssicht, vererben (Wiederverwenden) von Verhalten.
  - Zwischen Interfaces: Ein Interface erweitert die Menge der Operationen eines anderen Interfaces. Konzeptionell, Typing/Sub-Typing.



Klassendiagramm

- **Realisierung**:
  Beziehung zwischen einem Interface und einer Klasse. Die öffentliche Schnittstelle der Klasse ist konform zu den Operationen des Interfaces. „Vererben“ von Typ-Informationen, Typ-Konformität.




Entwurf für Converter UKR <==> BON / RBL, Version 6
Kommuniziert mit dem UKR-Bahnsystem
UDPServer
{singleton}
- instance: UDPServer {static}
- UDPServer()
+ getInstance():UDPServer {static}
+ start():void
+ stop():void
Ordnet Fahrzeugnummern Handles zu
HandleManager
{singleton}
- instance: HandleManager {static}
- HandleManager()
+ getInstance():HandleManager {static}
+ getHandle(vehicleID:int):int
VehicleInformation
{singleton}
- instance: VehicleInformation {static}
- handle: HashMap
- address: HashMap
- vehicleID: HashMap
- VehicleInformation()
+ getInstance():VehicleInformation {static}
+ getVehicleData(address:InetAddress):VehicleData
+ getVehicleData(handle:int):VehicleData
+ getVehicleData(vehicleID:int):VehicleData
+ setVehicleData(data:VehicleData):void
Informationen für ein Fahrzeug
VehicleData
- address: InetAddress
- handle: int
- vehicleID: int
+ VehicleData(vehicleID:int,handle:int,address:InetAddress)
+ VehicleData()
+ getVehicleID():int
+ getHandle():int
+ getAddress():string
+ setVehicleID(id:string):void
+ setHandle(handle:int):void
+ setAddress(address:string):void
Verallgemeinert die Konverter für TelegramConverter
<<interface>>
IConverter
+ convert(from:ITelegramContent,to:ITelegram):ITelegramContent
Kann ??? <=> ??? umwandeln
XYZTelegramConverter
+ XYZTelegramConverter()
+ convert(from:ITelegramContent,to:ITelegram):ITelegramContent
Kann UKR <=> BON umwandeln
UKRBONTelegramConverter
+ UkrBONTelegramConverter()
+ convert(from:ITelegramContent,to:ITelegram):ITelegramContent
Wandelt (GPS-)Koordinaten um
CoordinateTransformation
{singleton}
- instance: CoordinateTransformation {static}
- CoordinateTransformation()
+ getInstance():CoordinateTransformation {static}
Welche Koordinatensysteme kennen wir?
<<enum>>
CoordinateType
+ LONGITUDE_LATITUDE_DEGREE: CoordinateType {static}
+ LONGITUDE_LATITUDE_SEXAGESIMAL_MILLISECOND: CoordinateType {static}
[...]
- CoordinateType()
- CoordinateType(name:string)
+ getName():string
+ toString():string
+ getOrdinal():int
Kann alle Telegramme in jedes System umwandeln
TelegramConverter
{singleton}
- instance: TelegramConverter {static}
- TelegramConverter()
+ getInstance():TelegramConverter {static}
+ convert(from:ITelegramContent,to:ITelegram):ITelegramContent
Repräsentiert ein ganzes Telegramm
TelegramBON
- LOG: Logger {static}
- HELP: ConverterUtils {static}
- data: string
- fields: String[]
- type: BONTelegram=BONTelegram.NULL
+ TelegramBON(data:string)
+ TelegramBON(data:HashMap)
+ getITelegram():ITelegram
+ getTelegramName():string
+ getFieldContent(field:IFieldName):Object
+ getFieldString(field:IFieldName):string
+ getFieldInteger(field:IFieldName):int
+ toPeerMessage():PeerMessage
- analyseData():void
Abstrahiert ein Telegramm
<<interface>>
ITelegramContent
+ getITelegram():ITelegram
+ getTelegramName():string
+ getFieldContent(field:IFieldName):Object
+ getFieldString(field:IFieldName):string
+ getFieldInteger(field:IFieldName):int
<<interface>>
ITelegram
+ getName():string
+ toString():string
+ getOrdinal():int
Welche Telegramme gibt es?
<<enum>>
BONTelegram
- LOG: Logger
+ STANDORTMELDUNG_V13: BONTelegram {static}
+ STANDORTMELDUNG_V12: BONTelegram {static}
+ NICHT_ZUSTELLBAR_V10: BONTelegram {static}
+ FAHRTDATEN_V11: BONTelegram {static}
+ QUITTUNG_V10: BONTelegram {static}
+ NULL: BONTelegram {static}
+ OTHER: BONTelegram {static}
[...]
- BONTelegram()
- BONTelegram(name:string)
+ getName():string
+ toString():string
+ getOrdinal():int
<<enum>>
UKRTelegram
- LOG: Logger {static}
+ ACKNOWLEDGE: UKRTelegram {static}
+ SIGN_IN: UKRTelegram {static}
+ NULL: UKRTelegram {static}
+ OTHER: UKRTelegram {static}
[...]
- UKRTelegram()
- UKRTelegram(name:string)
+ getName():string
+ toString():string
+ getOrdinal():int
Repräsentiert ein ganzes Telegram
TelegramUKR
- LOG: Logger {static}
- HELP: ConverterUtils {static}
- data: byte[]
- type: UKRTelegram=UKRTelegram.NULL
+ TelegramUKR(data:byte[])
+ TelegramUKR(data:HashMap)
+ getITelegram():ITelegram
+ getTelegramName():string
+ getFieldContent(field:IFieldName):Object
+ getFieldString(field:IFieldName):string
+ getFieldInteger(field:IFieldName):int
Welche Felder gibt es?
<<enum>>
UKRFieldName
- LOG: Logger {static}
+ MsgID: UKRFieldName {static}
+ RepCount: UKRFieldName {static}
+ PacketID: UKRFieldName {static}
+ Handle: UKRFieldName {static}
+ Breitengrad: UKRFieldName {static}
+ OTHER: UKRFieldName {static}
+ NULL: UKRFieldName {static}
[...]
- UKRFieldName()
- UKRFieldName(name:string)
+ getName():string
+ toString():string
+ getOrdinal():int
+ getPosition():int
Welche Felder gibt es?
<<enum>>
BONFieldName
- LOG: Logger {static}
+ TeleType: BONFieldName {static}
+ TeleVersion: BONFieldName {static}
+ TeleID: BONFieldName {static}
+ GPSCoord1: BONFieldName {static}
+ OTHER: BONFieldName {static}
+ NULL: BONFieldName {static}
[...]
- BONFieldName()
- BONFieldName(name:string)
+ getName():string
+ toString():string
+ getOrdinal():int
+ getPosition():int
<<interface>>
IFieldName
+ getName():string
+ toString():string
+ getOrdinal():int
+ getPosition():int
Diverse Tools für kleine Umrechnungen
ConverterUtils
{singleton}
- instance: ConverterUtils {static}
- ConverterUtils()
+ getInstance():ConverterUtils {static}
+ makeInteger(binary:byte[]):int
+ readTextFile(file:string):String[]
+ makeBinary(value:int,countBytes:int):byte[]
+ getFields(data:string,regex:string):string[]
<<uses>>
<<implements>>

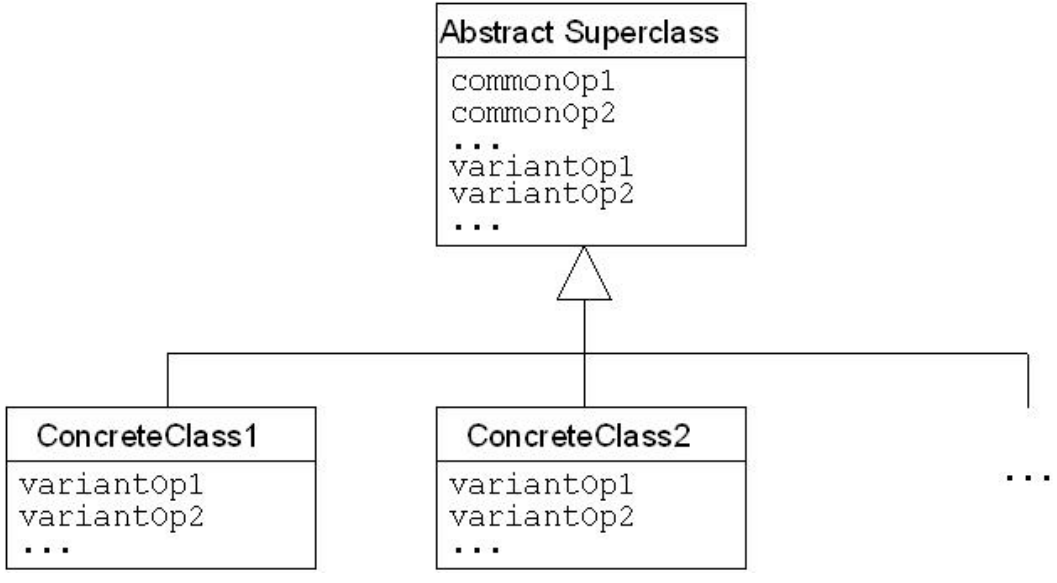
OO-Techniken und Design-Patterns

Agenda

- OO-Techniken und Designpatterns
  - Abstrakte Klassen, Interfaces und deren Kombination
  - Template Method
  - Iterator
  - Adapter
  - Null-Object
  - Composite
  - Decorator
  - Strategy

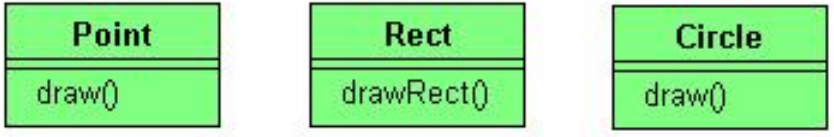
Abstrakte Basisklassen

- Fortführung von OO-Design-Gedanken der Sammlung von Funktionalität in Oberklassen
- Generalisierung erzeugt evtl. Basisklasse, die selbst nicht mehr sinnvoll instanziiert werden kann, da ihr Modellierungsdetails fehlen

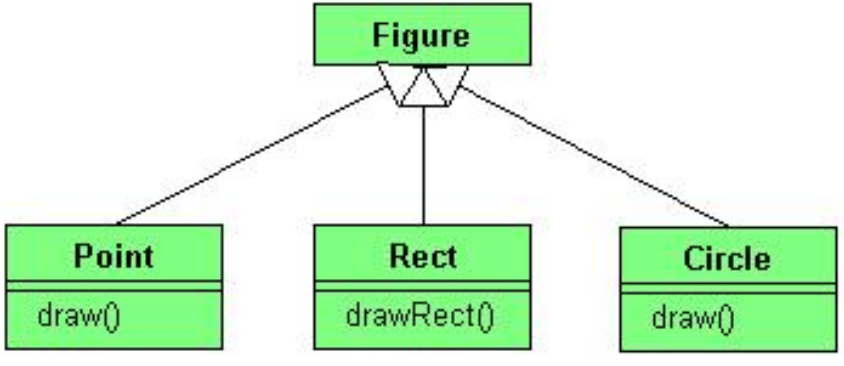


Abstrakte Basisklassen – Beispiel

- Beispiel: Grafische Figuren
  - Figuren = Klassen mit Methoden, um sich zu zeichnen



  - Figuren sollen einheitlich behandelt werden:
    - Schritt 1: Einführen einer Basisklasse

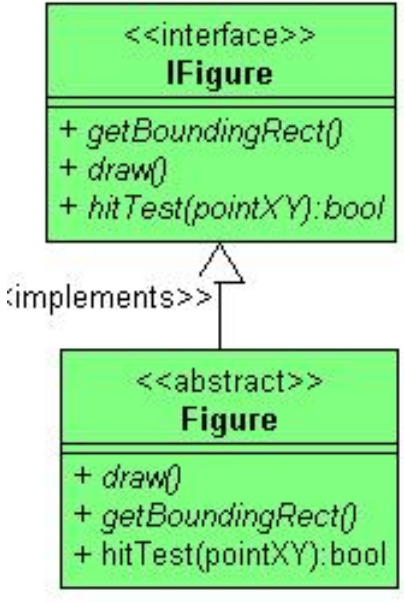


Abstrakte Basisklassen

- Vor- und Nachteile
  - [+] Konsistente Implementierung der Funktionalitäten, Subklassen bringen nur an speziellen, gewünschten Stellen ihre Realisierungen ein.
  - [+] Vermeidet Code-Duplikation und Wartungsaufwand
  - [+] Erzeugen neuer Subklassen ist einfacher
  - [-] Vielfach ist Ableitung nicht der richtige Designweg

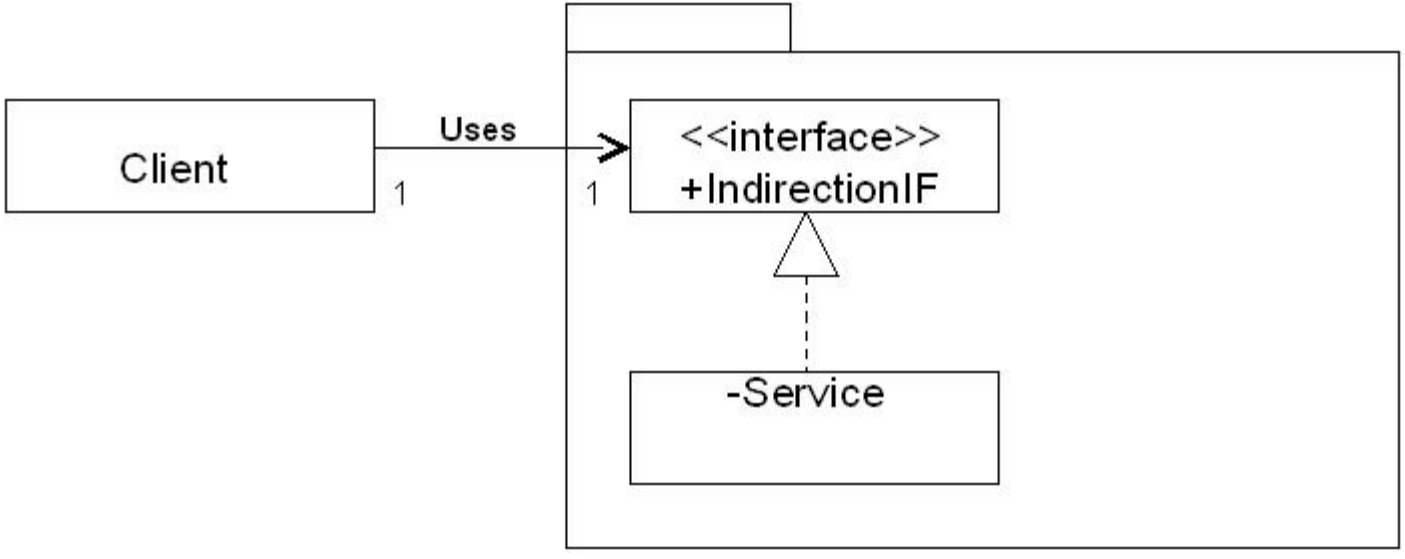
Interfaces

- Interfaces = Beschreibung, welche Methoden Klassen anbieten



Interfaces

- „Vertrag zwischen aufrufender und bereitstellender Klasse“



- Entkopplung von Realisierung und Spezifikation
- Austauschen verschiedener Implementierungen einfach möglich
  => **Factory Method-Pattern**

Interfaces

- Vor- und Nachteile
  - [+] Wiederverwendbarkeit einer Klasse ist höher, wenn andere Klassen sich nicht auf spezielle Implementierungen oder Implementierungsdetails verlassen
  - [+] konkrete Implementierung kann einfach ausgetauscht werden, ohne andere Objekte zu beeinträchtigen.
  - [o] kein Zugriff auf den Konstruktor

Interface + abstrakte Basisklasse

- Techniken Interface und abstrakte Basisklasse gut kombinierbar
- Vereint die Vorteile beider Techniken
  - Trennung von Spezifikation und Realisierung (Interface)
  - Sinnvolle Vorgabe von Basisfunktionalität (abstrakte Basisklasse)
  - Subklassen bringen an speziellen, gewünschten Stellen ihre Realisierungen ein

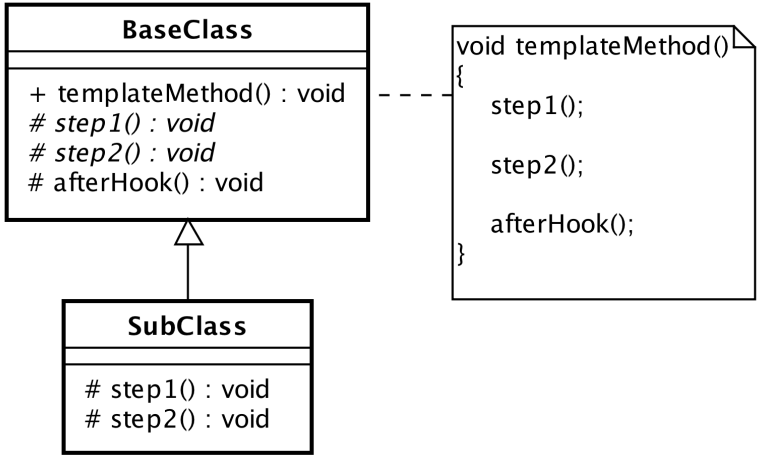
Template Method

Template Method

- **Template-Methode-Pattern definiert Grundzüge eines Algorithmus und erlaubt es Subklassen einige der Berechnungsschritte zu implementieren.**
- Idee ist es, den Algorithmus in **verschiedene Schritte** aufzuteilen
- Einige **Schritte** sind in Basisklasse noch **undefiniert** und können dann von Subklassen ausformuliert werden.
- **Realisierung durch abstrakte Methoden**
- Algorithmus vor Veränderungen zu schützen, kann die Template-Methode als `final` definiert werden

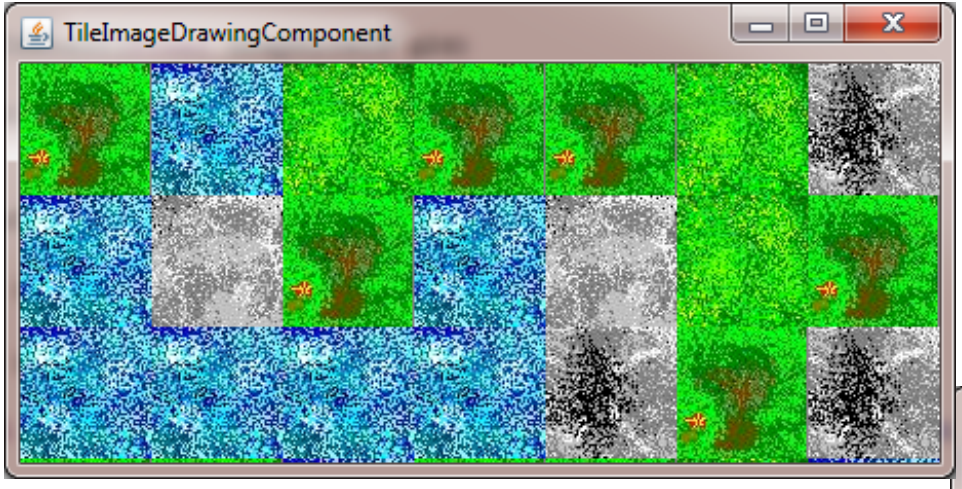
Template Method

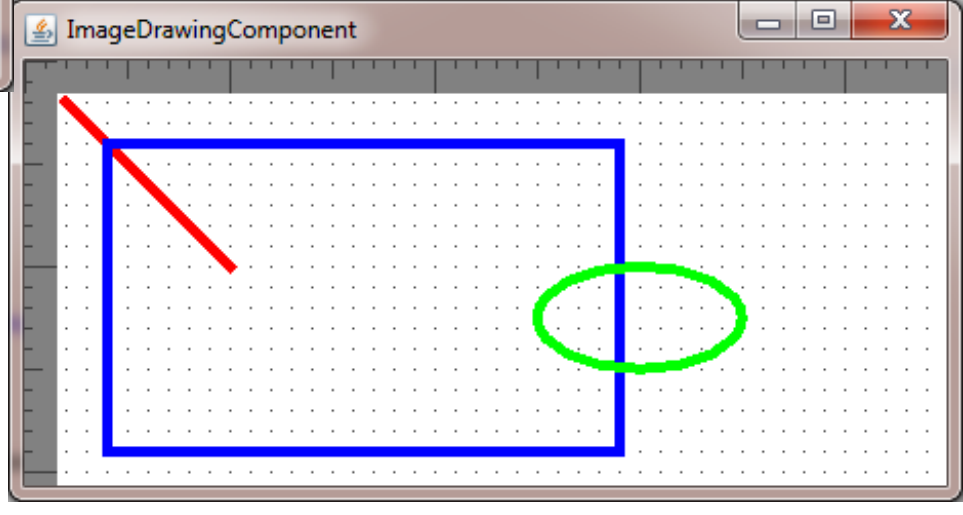
- **UML und Source-Code-Beispiel**




```
public void paint(Graphics g)
{
    super.paint(g);

    final Graphics2D g2d = (Graphics2D) g;

    drawSheetAndBackground(g2d);

    if (showRuler)
         drawRuler(g2d);

    if (showGrid)
         drawGrid(g2d);

    drawContent(g2d);
}

public abstract drawContent(Graphics2D g2d);
```

Template Method





Iterator

Iterator

- **Möglichkeit verschiedene Datenstrukturen auf gleiche Art zu durchlaufen**

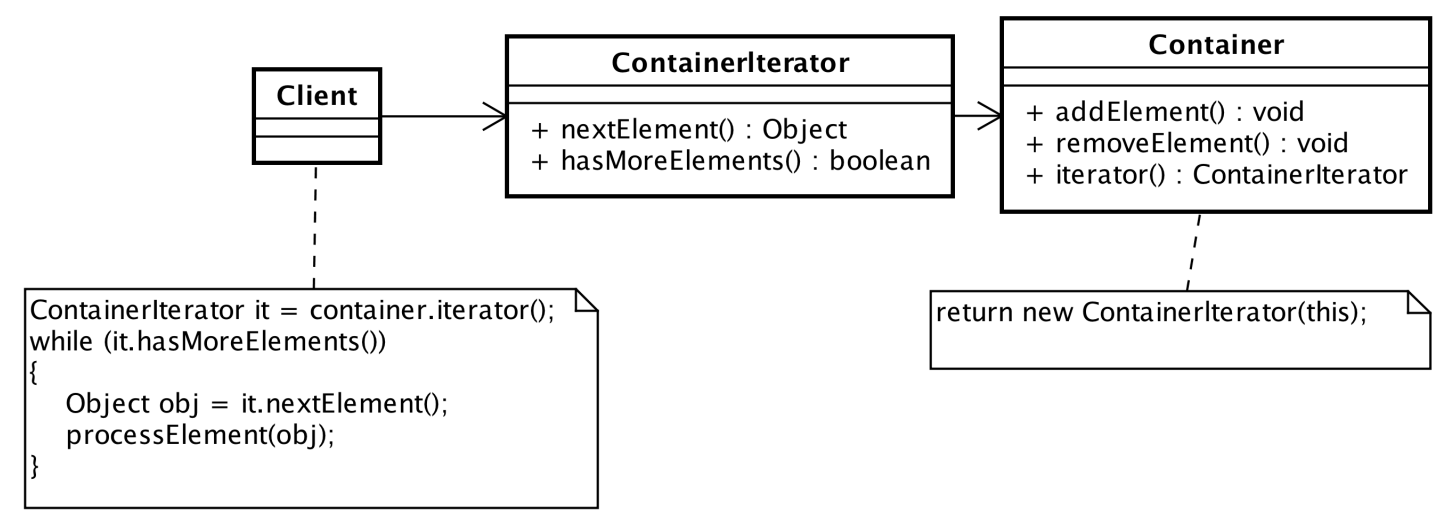
- **Abstraktion von der Datenstruktur**:
  - Keine Kenntnis des internen Aufbaus nötig
  - Bäume lassen sich durchlaufen wie Listen oder Sets

- **Traversierer wird Iterator genannt**

- **Mindestfunktionalität:**
  - nächste Element abfragen
  - Test auf weitere Elemente

Iterator – Beispiel

- `ContainerIterator` trennt `Container`-Objekt vom Traversierungsmechanismus
- Separation Of Concerns, Trennung der Zuständigkeiten:
  - Datenstruktur verwaltet nur die Daten,
  - jedoch **nicht die Art, wie sie durchlaufen wird.**

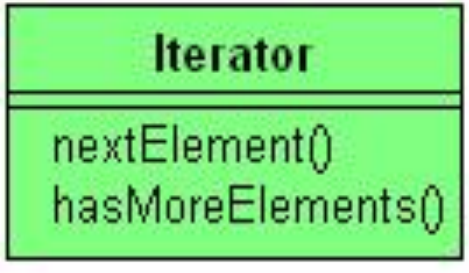


Iterator

- Probleme: Werden mehrere eigene Datenstrukturen erzeugt, so wäre es unpraktisch, dafür jeweils spezielle Iteratoren als Anwender kennen zu müssen. Erschwert Änderungen der Aggregationsklasse, da der Klientencode auf den speziellen Iteratortyp verweist.

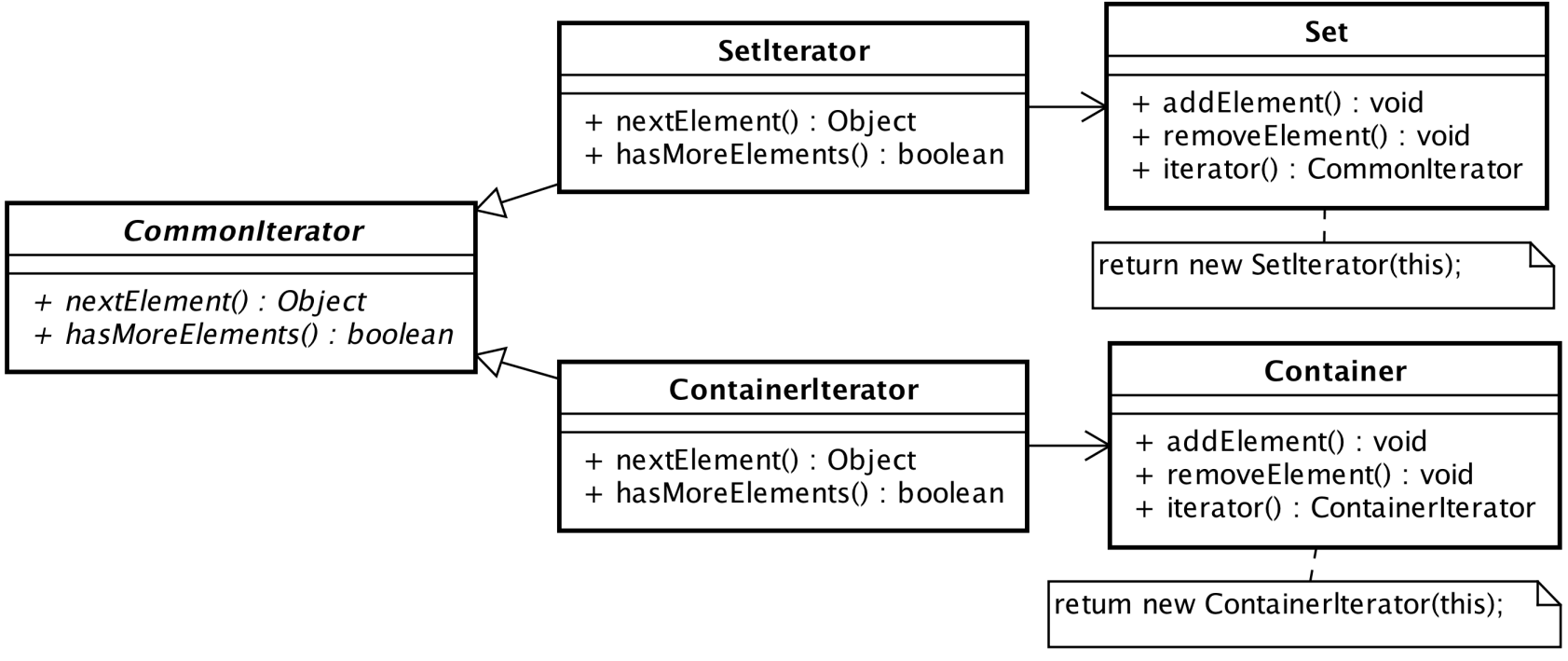
- Wir nutzen die beim OO-Design besprochene Technik des gemeinsamen Interfaces aus und definieren einen **polymorphen Iterator mit Interface Iterator als Basis**

- **Interface „Iterator" definiert eine Schnittstelle mit Methoden zur Traversierung.**



Iterator

- **Beispiel: Polymorpher Iterator**



Iterator im JDK

- Massiver Einsatz im Collections-Framework
- Alle dort definierten Collection lassen sich per Iterator traversieren.

Performance und Nebenbemerkungen

- Minimalen Einfluss auf die Performance
- Lesbarkeit und Abstraktion viel besser als bei einer for-Schleife
- Java 5 bietet for-each-Schleife um die Lesbarkeit zu erhöhen:

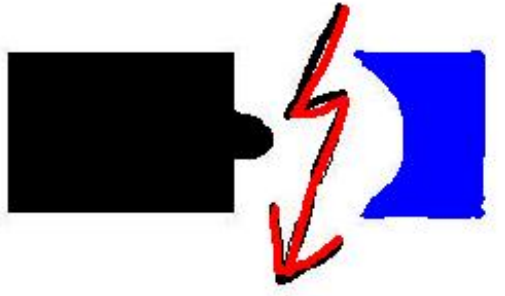
```
for (TimerTask t : collection)
     t.cancel();
```

- Allerdings nicht die Möglichkeiten des Iterators
  - `remove()` oder
  - Spezialbehandlungen bei einzelnen Elementen

Adapter

Adapter

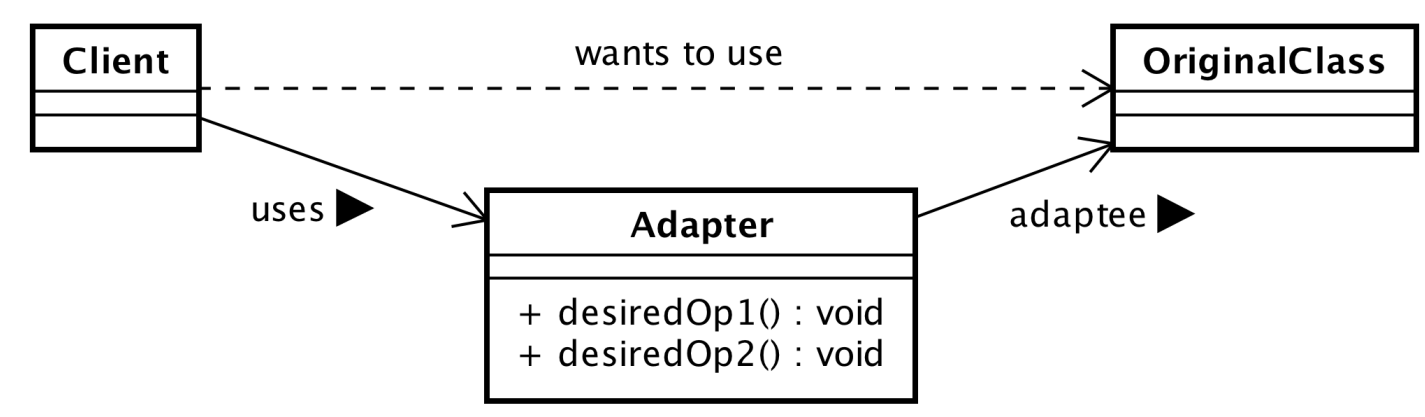
- ansonsten inkompatible Software kompatibel gemacht werden kann



- Mithilfe von Interfaces und dem vorgestellten Adapter-Pattern können Klassen kompatibel gemacht werden

Adapter

- Ein Klient Client nutzt ein Objekt vom Typ Adapter, um die Funktionalität einer vorhandenen Klasse OriginalClass mit einer für den Klienten passenden Schnittstelle verwenden zu können.



Adapter Beispiel

- Eine List<T> mit Swing-ListModel kompatibel werden: List -> ListModel

```
public static class ListToListModelAdapter<E> extends AbstractListModel<E>
{
    private final List<E> adaptee = new ArrayList<>();

    public ListToListModelAdapter(final List<E> data)
    {
        this.adaptee.addAll(data);
    }

    public int getSize()
    {
        return adaptee.size();
    }

    public E getElementAt(final int i)
    {
        return adaptee.get(i);
    }
}
```

Adapter

- Problem: Wir haben ein String[ ] und wollen es per Iterator durchlaufen
- In Java nicht möglich, da Arrays keine Iteratoren bereitstellen

```
final String[] names = { "first", "second", "last" };

// conventional loop
for ( int i = 0; i < names.length;i++)
{
   final String currentName = names[i];
   System.out.println("Name = '" + currentName +"'.");
}
```

- Wir wollen aber bevorzugt Iteratoren einsetzen, also Adapter einsetzen!

```
final String[] names = { "first", "second", "last" };

final Iterator it = ???? Names ????
while ( it.hasNext() )
{
   final String currentName = (String)it.next();
   System.out.println("Name = '" + currentName +"'.");
}
```

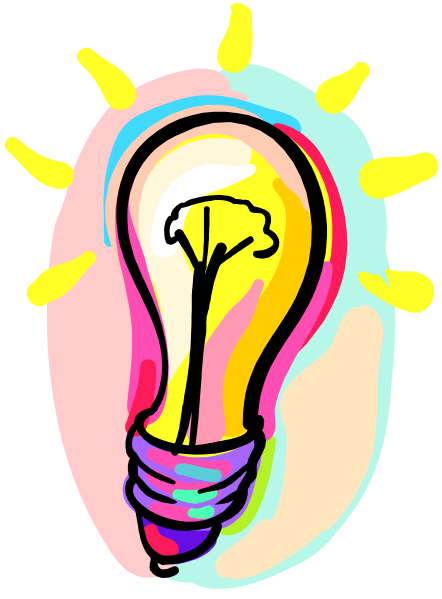
Adapter – Übung

Nutze das Adapter-Pattern, um eine Möglichkeit zu bieten,

Arrays mit einem Iterator zu durchlaufen.

```
public class ArrayIteratorAdapter implements Iterator
{
    int currentPosition;
    final Object[] adaptee;

    public ArrayIteratorAdapter(final Object[] adaptee)
    {
          this.adaptee = adaptee;
          this.currentPosition = 0;
    }

    // TODO
    public boolean hasNext()
    public Object next()
    public void remove()
}
```

Adapter – Beispiellösung



```
public class ArrayIteratorAdapter implements Iterator
{
    int currentPosition;
    final Object[] adaptee;

    public ArrayIteratorAdapter(final Object[] adaptee)
    {
          this.adaptee = adaptee;
          this.currentPosition = 0;
    }

    public boolean hasNext()
    {
          return currentPosition < adaptee.length;
    }

    public Object next()
    {
          final Object next = adaptee[currentPosition];
          currentPosition++;
          return next;
    }

    // Da wir nur einen Wrapper darstellen, dürfen und
    wollen wir nicht verändern!!!
    public void remove()
    {
          throw new UnsupportedOperationException("Adapter
    does not implement remove!");
    }
}
```

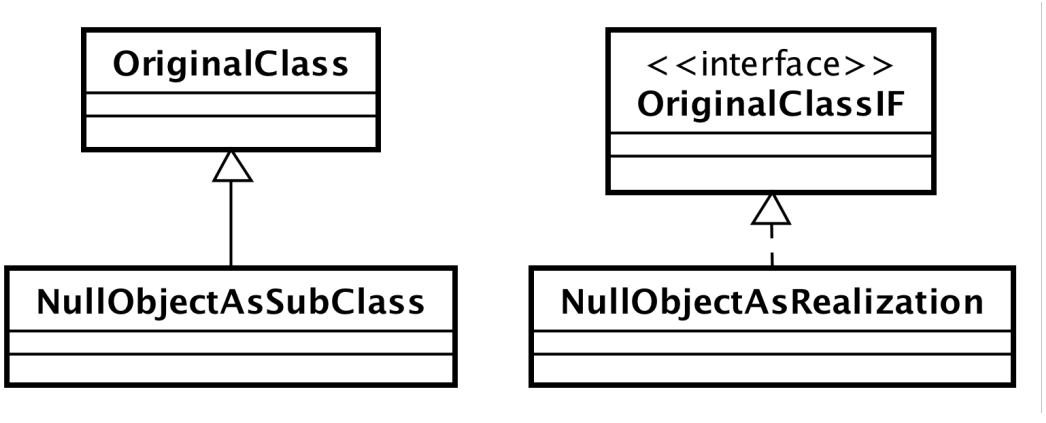
Null Object

Null Object

- **Idee des Null-Objects = Platzhalter für `null`-Referenzen**
- Modelliert ein nicht vorhandenes Objekt.
- Einsatz dort, wo ansonsten `null` verwendet oder übergeben wird
- Die Realisierung erfolgt in der Regel durch Vererbung, zum Teil auch durch Implementierung eines Interfaces: **Alle Methoden „leer" implementiert**.
- Beispiele für das Null-Object-Pattern findet man im Collections-Framework mit den Klassen `EMPTY_LIST`, `EMPTY_SET`, usw.

Null Object

- Um als »funktionsloser« Ersatz dienen zu können, muss ein Null-Objekt die Schnitt- stelle der zu vertretenden Klasse erfüllen.
- Die Implementierung erfolgt entweder durch Vererbung oder durch Realisierung eines Interface.



Null Object – Tipps zur Realisierung

- `void`-Methoden werden leer implementiert.
- Methoden mit Rückgabewert erfordern etwas mehr Aufwand:
  - `boolean`-Methoden geben häufig `false` zurück
  - Für `int` bieten sich `0` oder `-1` als Rückgabewerte an
    gleiches gilt für weitere Zahlentypen (`float`, `double`, usw.)
  - Objekt-Typen kann entweder rekursiv Null-Object-Pattern oder `null`
  - Für Collections verwenden wir die Konstanten aus dem Collections-Framwork: `EMPTY_LIST`, `EMPTY_SET` usw.
  - Arrays sollten ein leeres Array zurückgegeben

Null Object – Bewertung

- **Vorteile:**
  - **vereinfacht die gesamte Behandlung im Quelltext, da weder `null`-Checks noch Spezialbehandlungen einzubauen sind.**
  - **Quelltext wird kürzer und übersichtlicher**
  - **Quelltext deutlich lesbarer**
- **Alternative mit Java 8: Die Klasse Optional**

Null Object – Bewertung

- **Vorteile:**
  - **vereinfacht die gesamte Behandlung im Quelltext, da weder `null`-Checks noch Spezialbehandlungen einzubauen sind.**
  - **Quelltext wird kürzer und übersichtlicher**
  - **Quelltext deutlich lesbarer**
- **Alternative mit Java 8: Die Klasse Optional**

Null Object – Übung

Nutze das Null-Object-Pattern, um einen „Leer“-Iterator zu schreiben. Verwende wieder das Java Iterator-Interface als Basis!

```
public class NullIterator implements Iterator
{
    // TODO
    public boolean hasNext()
    public Object next()
    public void remove()
}
```

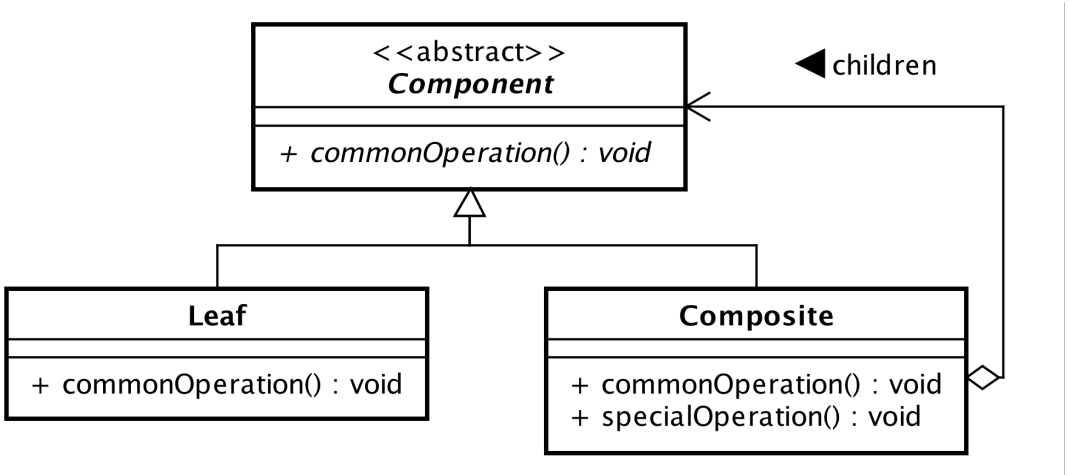
Null Object – Beispiellösung

```
public final class NullIterator implements Iterator
{
   public boolean hasNext()
   {
        return false;
   }

   public Object next()
   {
        throw new NoSuchElementException("No elements");
         // Alternative: return null;
   }

   public void remove()
   {
         throw new UnsupportedOperationException(
                    "NullIterator does not implement remove!");
   }
}
```

Composite

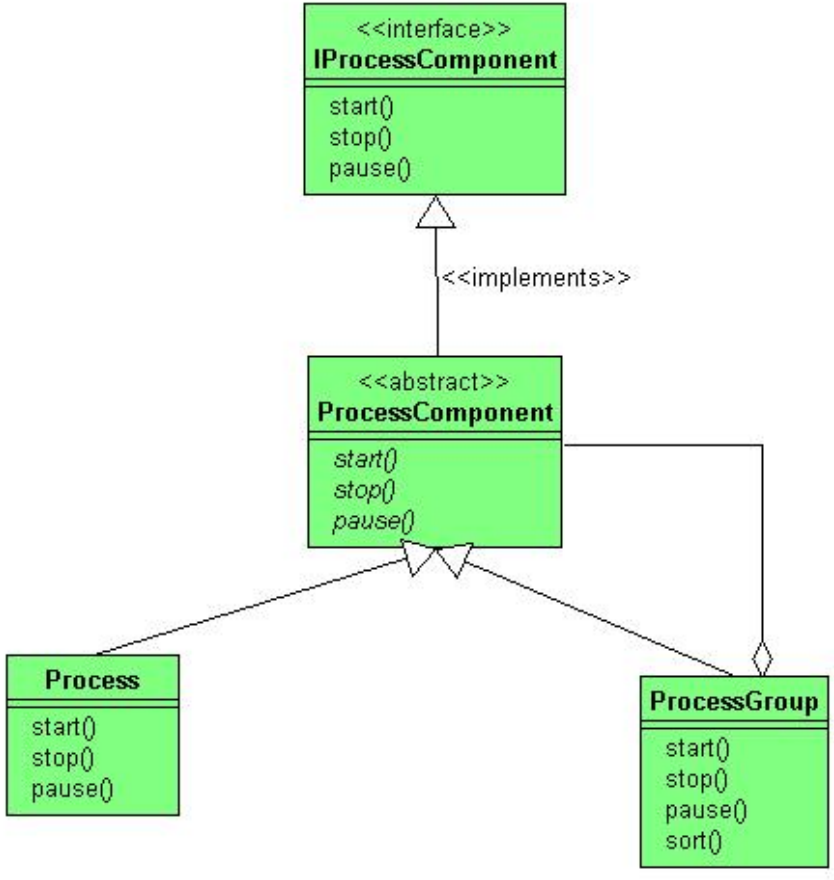
Composite

- **Ermögliche Elemente in Baum-artigen Strukturen gleich zu behandeln.**
- Ähnliche Ideen wie Iterator: Möglichkeit verschiedene Datenstrukturen auf gleiche Art zu durchlaufen; hier Behandlung der Elemente
- Keine Unterscheidung Composite/Container und Einzelelemente nötig
- **Spezialstruktur mit abstrakter Basisklasse wird Kompositum genannt**



Composite

- **Kombination mit der Technik Interface + Abstrakte Basisklasse**

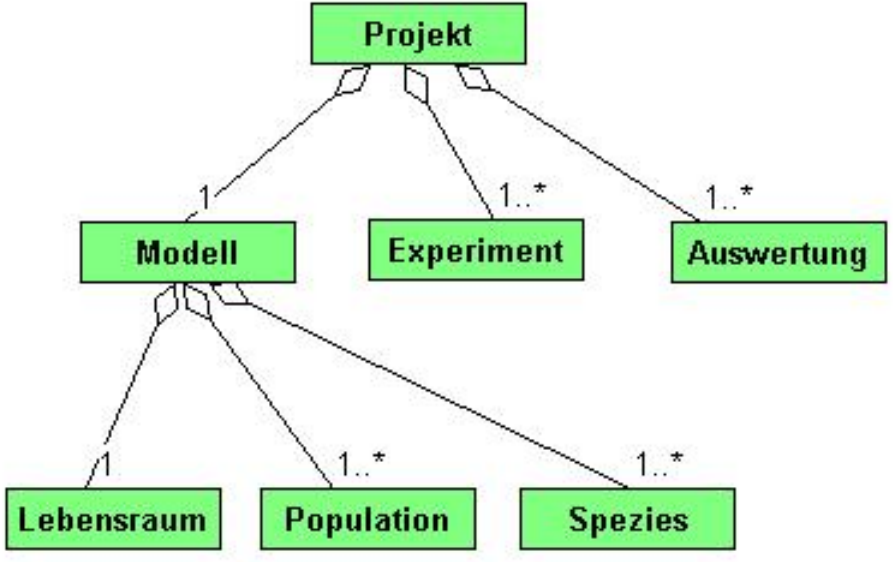


Composite – Übung

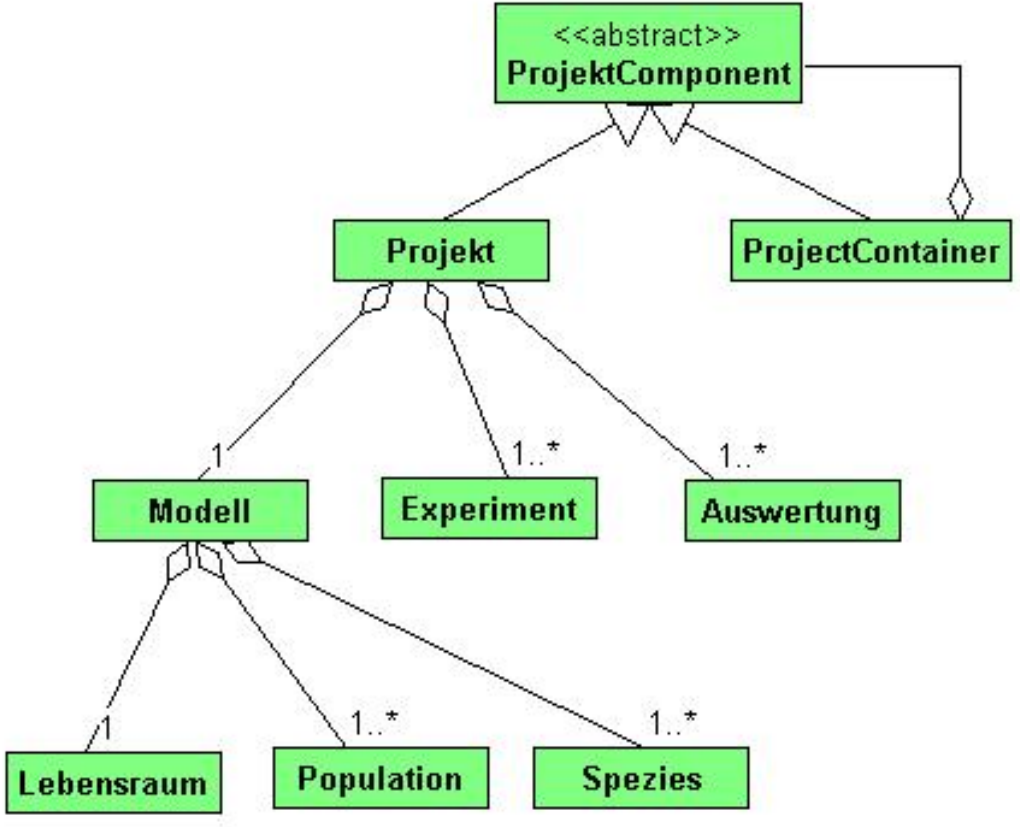
Modelliere eine Projektverwaltung mit UML:

1. **Ein Projekt besteht aus einem Modell und mehreren Experimenten und Auswertungen. Modelle enthalten Populationen, Spezies und einen Lebensraum.**
2. Ein Projekt kann beliebig viele Sub-Projekte enthalten! Verwende das Kompositum-Muster!

Composite – Beispiellösung Teil 1



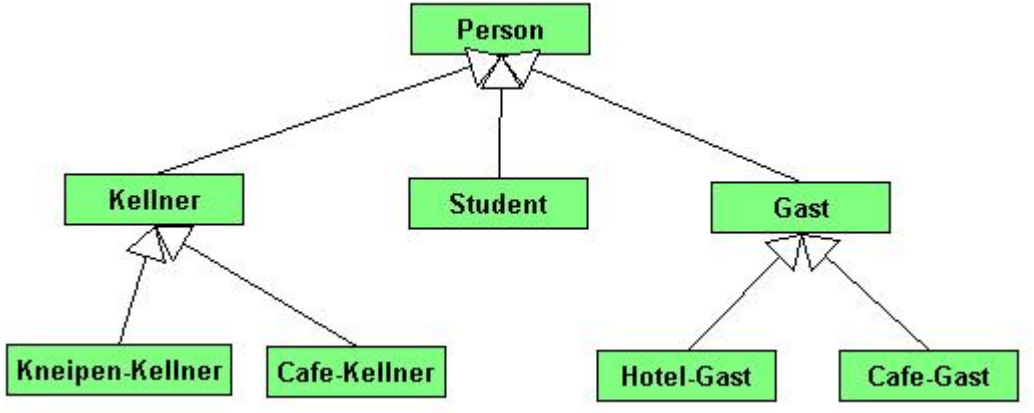
Composite – Beispiellösung Teil 2



Decorator

Probleme mit Vererbung

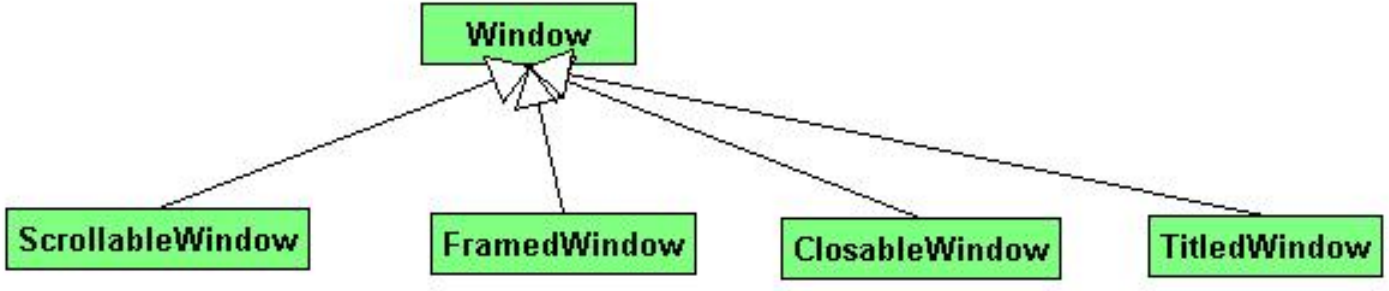
- **Selbst beim Einhalten der „is-a“-Eigenschaft kann es zu Problemen kommen:**



- **Sind alles Spezialisierungen**
- **ABER: Zielen mehr auf eine Rolle/Aufgabe, die zeitweilig ausgeübt wird**
  - **„is-a-role-played-by“ => Delegation**
  - **„can-act-like“ => Interface**

Probleme mit Vererbung

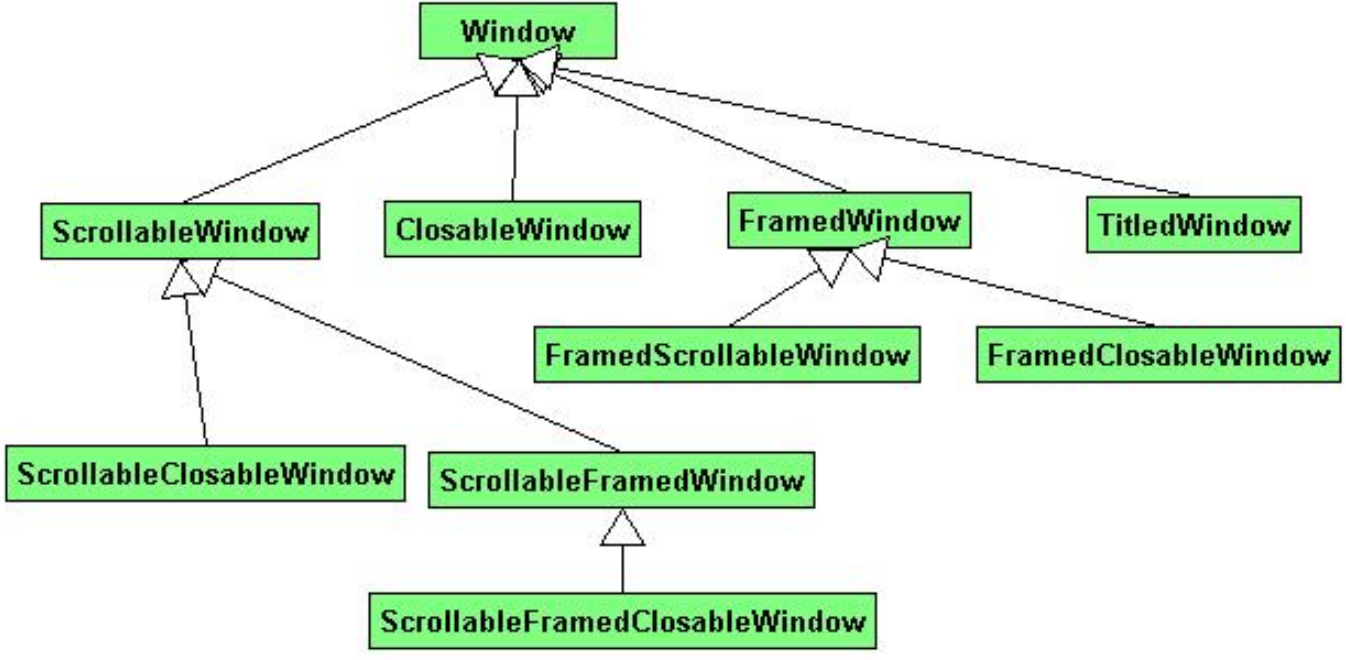
- Eigenschaften per Vererbung hinzufügen:



- Sind alles Spezialisierungen
- ABER: Beschreiben mehr eine (boolesche) Eigenschaft
- Problem: Man möchte die Eigenschaften kombinieren
  => weitere Ableitungen notwendig
  => Kombinatorische Explosion

Probleme mit Vererbung

- Kombinatorische Explosion:



- **Gibt es Unterschiede bei verschiedenen Ableitungsreihenfolgen?**
- **Für orthogonale (voneinander unabhängige) Eigenschaften: => Decorator-Pattern: Füge Funktionalität dynamisch ohne Vererbung hinzu**

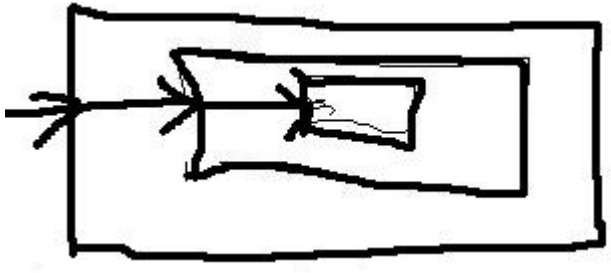
Decorator

- **zusätzliches Verhalten transparent zur Verfügung stellen**

- Keine Modifikation der ursprünglichen Klasse
- Keine Vererbung
- Neue Funktionalität zur Laufzeit, also dynamisch

=> Klingt ja unglaublich !? Wie geht das?

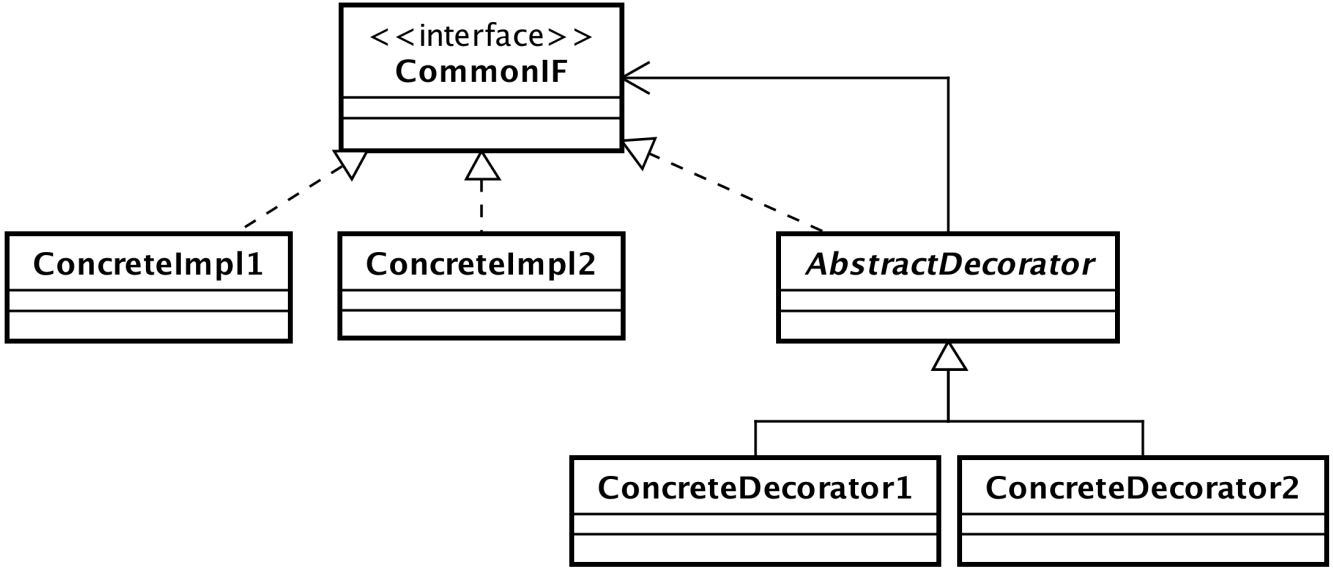
- Über das Erfüllen eines Interfaces, Ummanteln mit Funktionalität



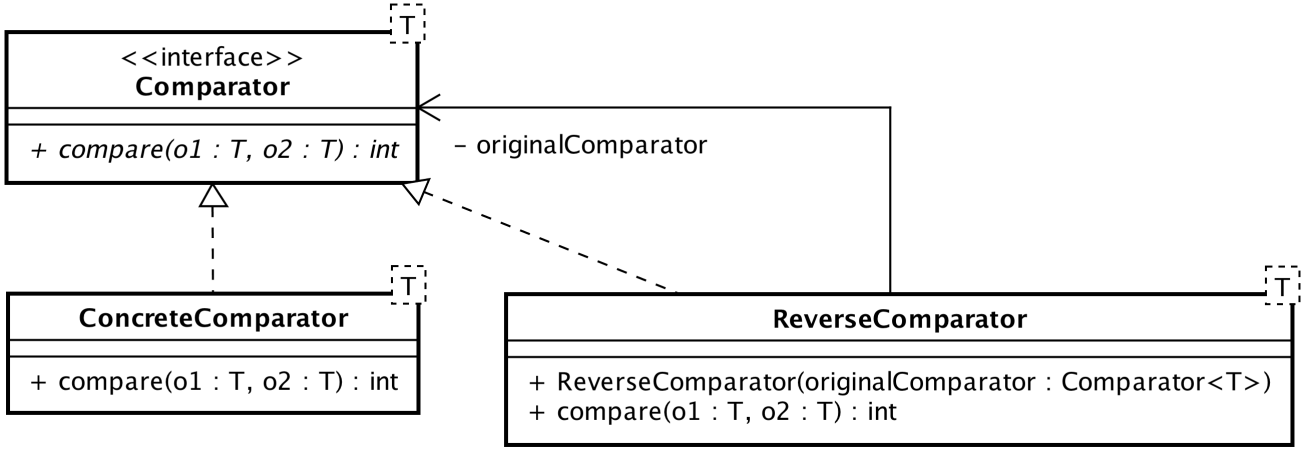
- Aber: Keine Kontrolle, welche Funktionalität hinzugefügt wird

Decorator Struktur

- gemeinsamer Basistyp (CommonIF)
- Bei Aufruf von Methoden eines Dekoriererobjekts ruft dieses gleichlautende Methoden des referenzierten, zu dekorierenden Objekts auf und delegiert somit diesen Auftrag
- Problemlos können auch mehrere Dekoriererobjekte hintereinander geschaltet werden, d. h., es findet dann eine mehrfache Ummantelung statt.



Decorator

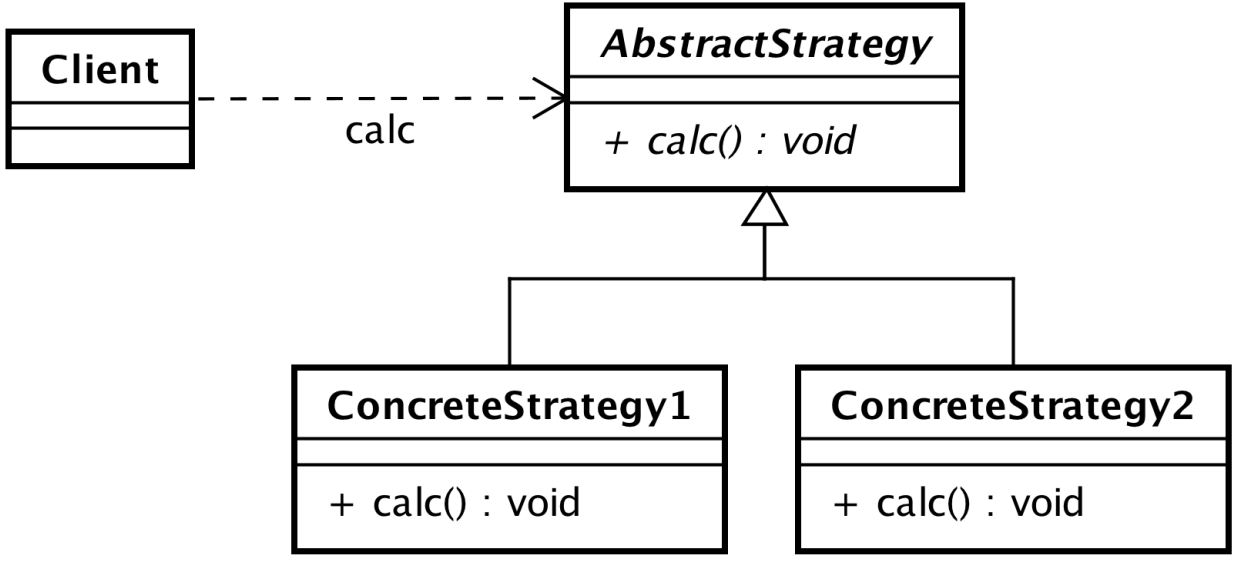



```
public final class ReverseComparator<T> implements Comparator<T>
{
    private final Comparator<T> originalComparator;

    public ReverseComparator(final Comparator<T> originalComparator)
    {
        this.originalComparator = Objects.requireNonNull(originalComparator,
                                    "originalComparator must not be null!");
    }

    @Override
    public int compare(final T o1, final T o2)
    {
        return originalComparator.compare(o2, o1);
    }
}
```

Strategy

Strategy

- **Verhalten eines Algorithmus an ausgesuchten Stellen anpassen.**
- variable Bestandteile eines Algorithmus werden durch eigene Klassen statt durch überschriebene Methoden realisiert.

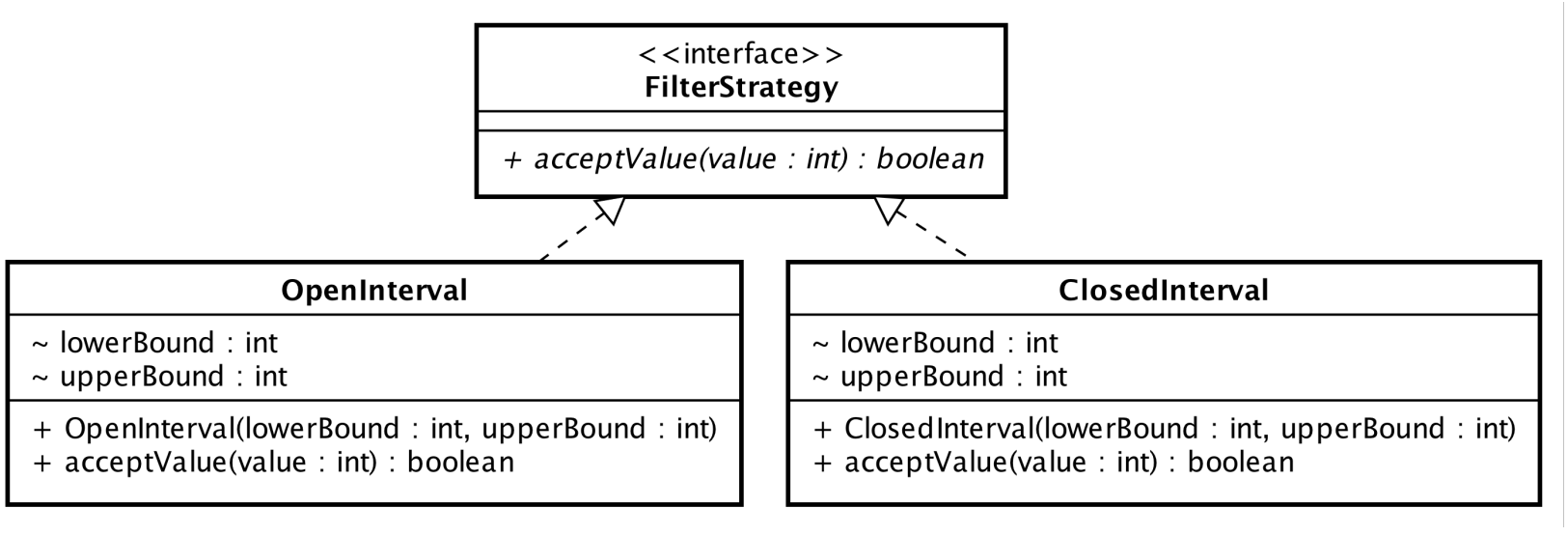


Strategy – Beispiel

```
public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterType filterStrategy,
                                      final int lowerBound,
                                      final int upperBound)
{
    final List<Integer> filteredList = new LinkedList<>();

    if (filterStrategy == FilterType.CLOSED_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value >= lowerBound && value <= upperBound)
                filteredList.add(value);
        }
    }
    if (filterStrategy == FilterType.OPEN_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value > lowerBound && value < upperBound)
                filteredList.add(value);
        }
    }

    return filteredList;
```

Strategy

- Die Basis bildet ein Interface `FilterStrategy` mit der dort definierten
- Methode `acceptValue(int)`. Zur Intervallprüfung werden zwei konkrete
- Strategieklassen `OpenInterval` und `ClosedInterval` definiert



Strategy

```
public static class ClosedInterval implements FilterStrategy
{
   private final int lowerBound;
   private final int upperBound;

   public ClosedInterval(final int lowerBound, final int upperBound)
   {
      if (upperBound < lowerBound)
         throw new IllegalArgumentException("lowerBound must be <= upperBound");

      this.lowerBound = lowerBound;
      this.upperBound = upperBound;
   }

   @Override
   public boolean acceptValue(final int value)
   {
       return lowerBound <= value && value <= upperBound;
   }

   @Override
   public String toString()
   {
      return "ClosedInterval [" + lowerBound + ", " + upperBound + "]";
   }
}
```

Strategy

- **Verhalten eines Algorithmus an ausgesuchten Stellen anpassen**
- variable Bestandteile durch `FilterStrategy` realisiert.

```
public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterStrategy filterStrategy)
{
   final List<Integer> result = new LinkedList<>();
   for (final Integer value : inputs)
   {
      if (filterStrategy.acceptValue(value))
         result.add(value);
   }
   return result;
}
```

Übungen Design Patterns

Literatur

Literatur

UML:

- Das UML Benutzerhandbuch,
  Booch, Rumbaugh, Jacobson, Addison-Wesley, 2006
- Objektorientierte Softwarentwicklung mit UML,
  Forbrig, Fachbuchverlag Leipzig, 2002

Entwurfsmuster:

- Head First Design Patterns,
  Freeman, Freeman, Sierra, Bates, O'Reilly, 2004
- Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software,
  Gamma, Helm, Johnson, Vlissides, Addison Wesley, 2004
- Design Patterns Java Workbook,
  Metsker, Addison-Wesley, 2002

Vielen Dank für die Teilnahme und happy coding!