

Objektorientierte Programmierung

Hochschule Bochum

WS 19/20

Dr.-Ing. Darius Malysiak

IV. Design Patterns

Unified Modeling Language (UML):

- **Grafische** Sprache zur Repräsentation von **Entitäten**
- Aktuelle Version 2.5 (2015)
- Geprägt von Grady Booch, Ivar Jacobson und James Rumbaugh
- Gegliedert in Spracheinheiten (Menge von häufig wechselwirkenden Komponenten / Language Units):
 - Interactions (z.B. für Sequenzdiagramme, Kommunikationsdiagrammen)
 - Classes (z.B. für Klassendiagramme, Objektdiagrammen)
 - State Machines (z.B. für Zustandsautomaten)
- Darstellung variiert entgegen des Standards!
- Wir behandeln nur Klassen-/Objektdiagramme

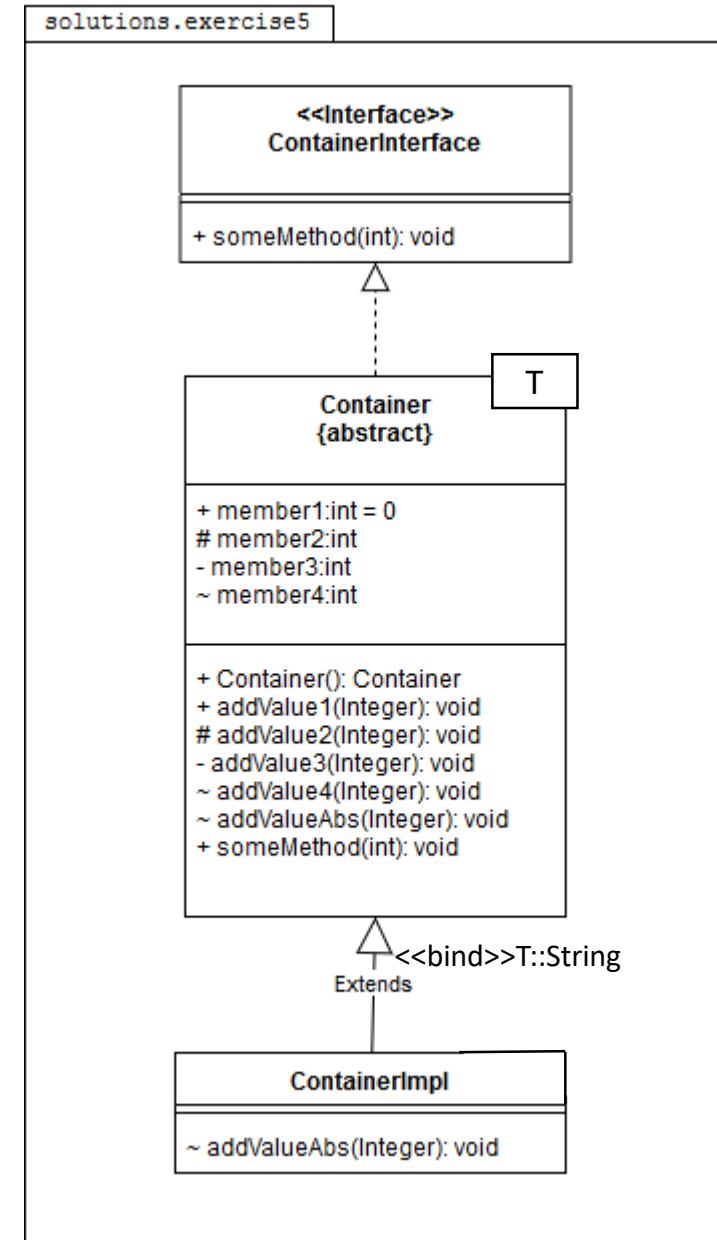
IV. Design Patterns

Unified Modeling Language (UML):

```
public interface ContainerInterface {  
    public void someMethod(int value);  
}
```

```
public abstract class Container<T>  
    implements ContainerInterface{  
    public int member1=0;  
    protected int member2;  
    private int member3;  
    int member4;  
  
    public Container() {}  
  
    @Deprecated  
    public void addValue1(Integer val) {}  
    protected void addValue2(Integer val) {}  
    private void addValue3(Integer val) {}  
    void addValue4(Integer val) {}  
    abstract void addValueAbs(Integer val);  
  
    @Override  
    public void someMethod(int value) {}  
}
```

```
public class ContainerImpl extends Container<String>{  
    @Override  
    void addValueAbs(Integer val) {}  
}
```

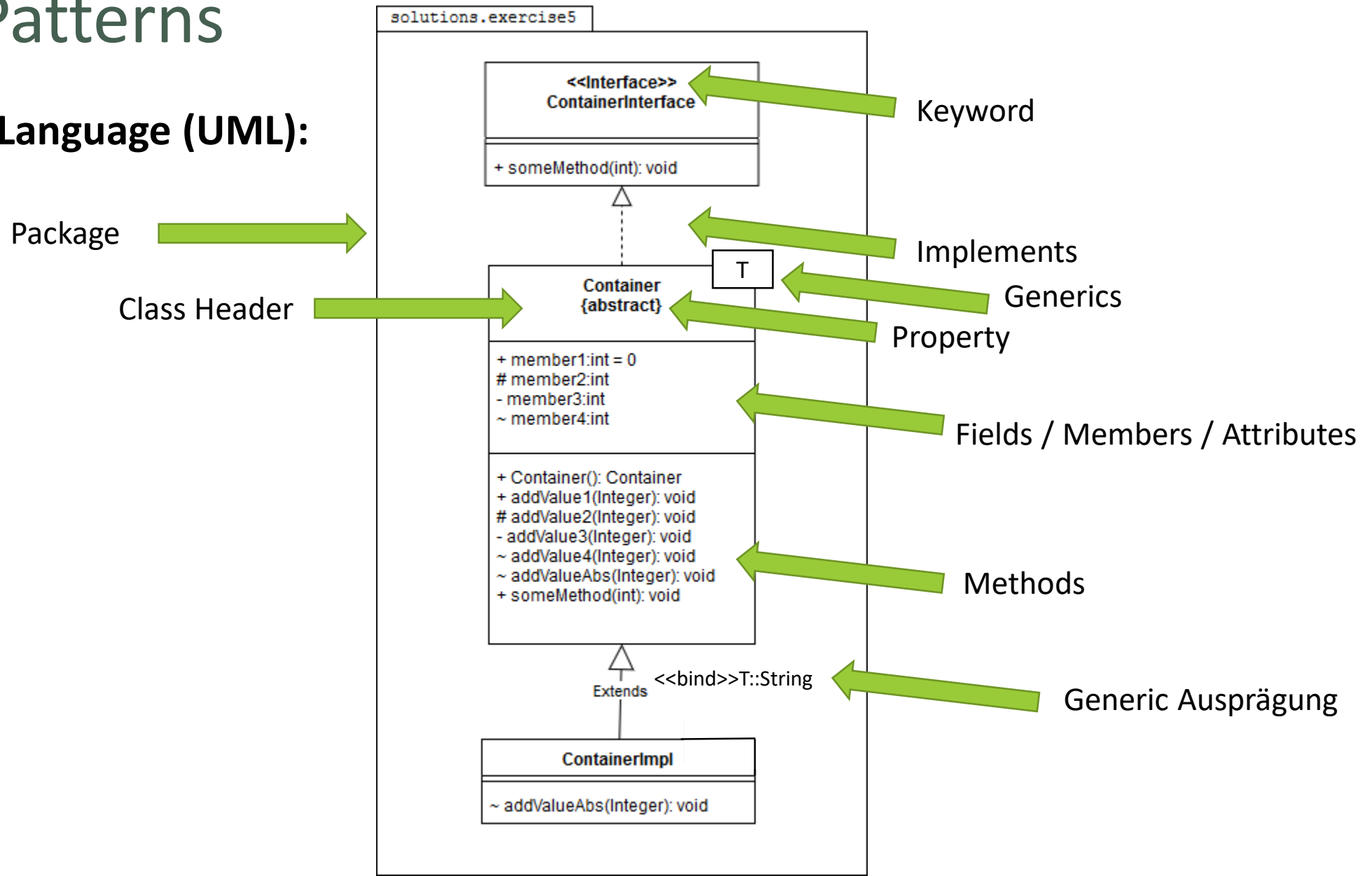


IV. Design Patterns

Unified Modeling Language (UML):

Modifier:

+ public
protected
- private
~ package



IV. Design Patterns

Unified Modeling Language (UML):

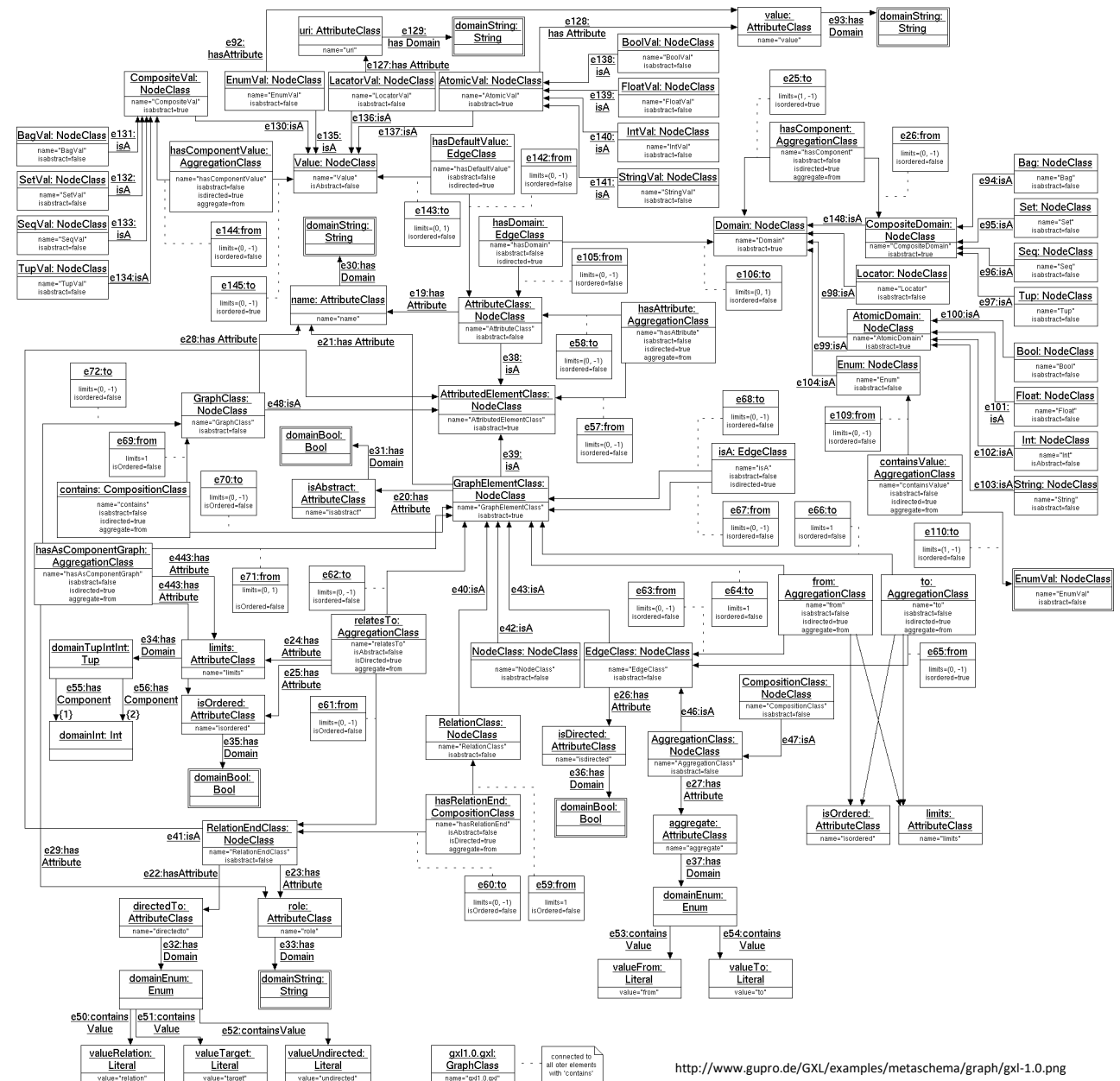
Was ist das Ziel von UML in der Praxis ?

- Grobe Übersicht der Klassen-Relationen.
- Grobe Übersicht der ObjektRelationen.

Business Logic wie z.B. Initialisierung von Attributen ist im Code zu sehen.

Häufig wird eine vereinfachte Notation genutzt:

- Weniger Details
- Selektives Ein-/Ausblenden



IV. Design Patterns

Unified Modeling Language (UML):

```
public interface ContainerInterface {  
    public void someMethod(int value);  
}
```

```
public abstract class Container<T>  
    implements ContainerInterface{  
    public int member1=0;  
    protected int member2;  
    private int member3;  
    int member4;  
  
    public Container() {}  
  
    @Deprecated  
    public void addValue1(Integer val) {}  
    protected void addValue2(Integer val) {}  
    private void addValue3(Integer val) {}  
    void addValue4(Integer val) {}  
    abstract void addValueAbs(Integer val);  
  
    @Override  
    public void someMethod(int value) {}  
}
```

```
public class ContainerImpl extends Container<String>{  
    @Override  
    void addValueAbs(Integer val) {}  
}
```



= public

= protected

= private

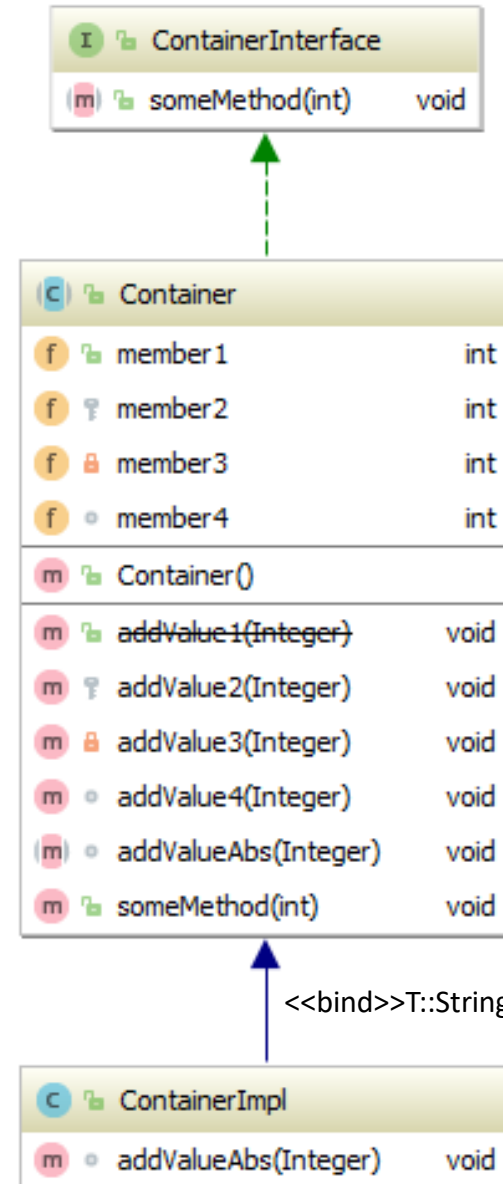
= package

= Class

= Interface

Implements

Extends



IV. Design Patterns

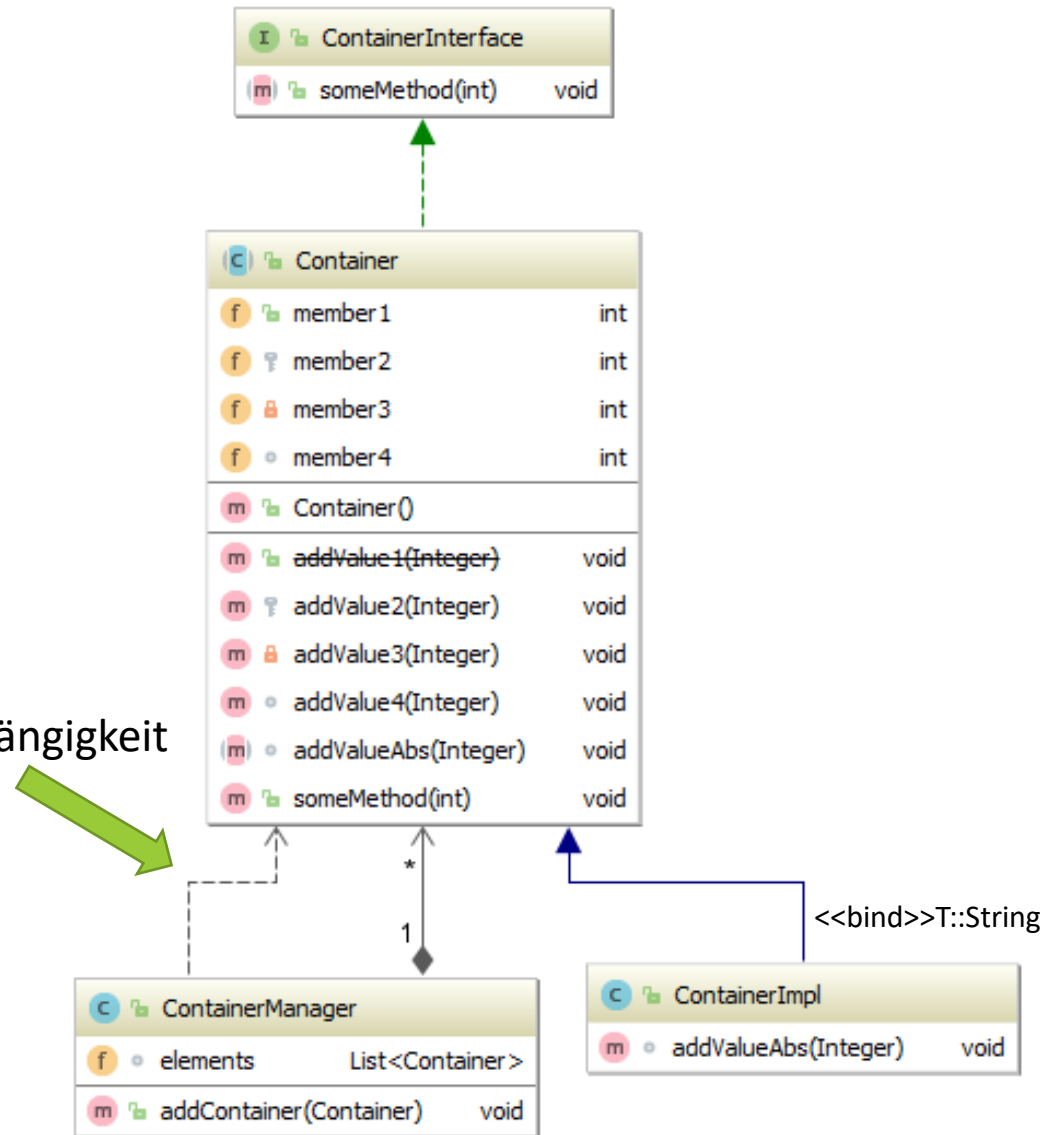
Unified Modeling Language (UML):

```
public class ContainerManager {  
  
    List<Container> elements = new ArrayList<>();  
  
    public void addContainer(Container c)  
    {  
        elements.add(c);  
    }  
}
```

Komposition:

- Eine Form Objektbeziehung.
- Mindestens ein Element der Zielklasse (Pfeil) wird in der Quellklasse (Raute) benötigt.
- „1 zu N“

Beziehung / Abhängigkeit



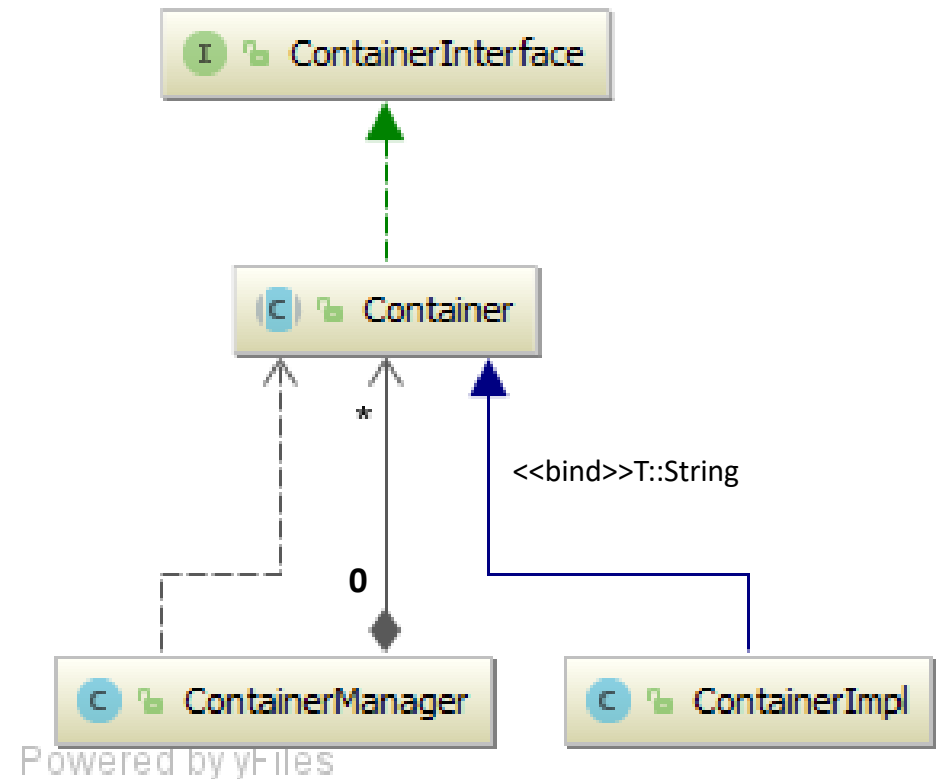
IV. Design Patterns

Unified Modeling Language (UML):

```
public class ContainerManager {  
  
    List<Container> elements = new ArrayList<>();  
  
    public void addContainer(Container c)  
    {  
        elements.add(c);  
    }  
}
```

Aggregation:

- Verallgemeinerung der Komposition.
- Es wird nicht unbedingt ein Element der Zielklasse benötigt.
- „0 zu N“



IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Sei $f : T \rightarrow \{true, false\}$ eine prüfbare Eigenschaft über T , so gilt:

$$S \subseteq T \wedge f(T) = \{true\} \wedge y \in S \Rightarrow f(y) = true$$

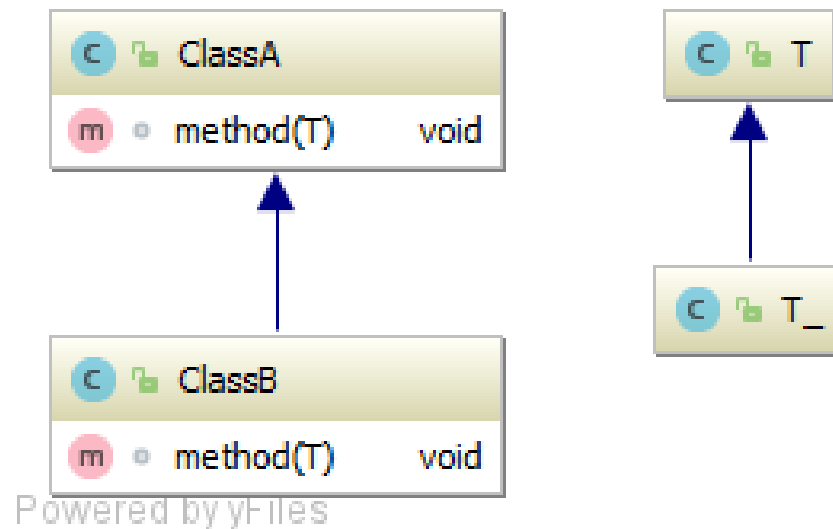
Dieses Prinzip hatte großen Einfluss auf die Entwicklung von Programmiersprachen und implizierte:

- Kovarianz
- Kontravarianz
- Bivarianz
- Invarianz

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Invarianz (Parameter)

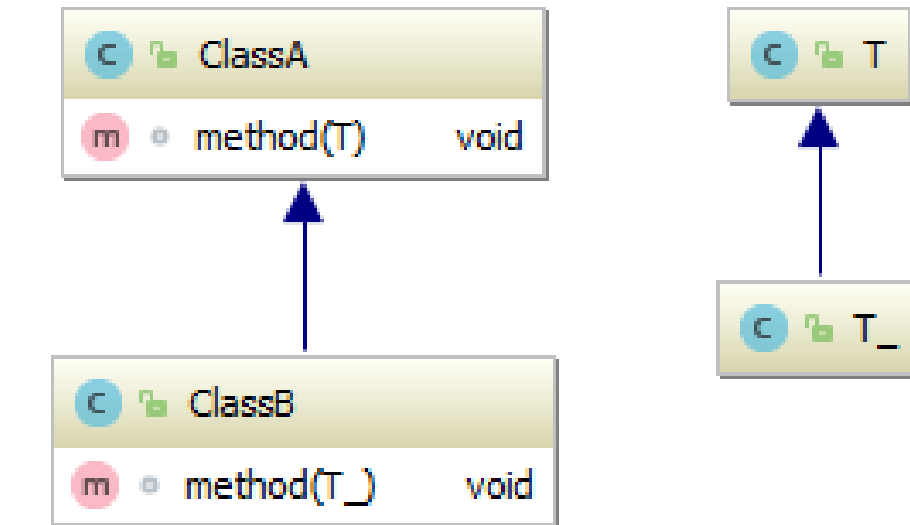


IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Kovarianz (Parameter)

„method“ wird nicht überschrieben
sondern überladen.



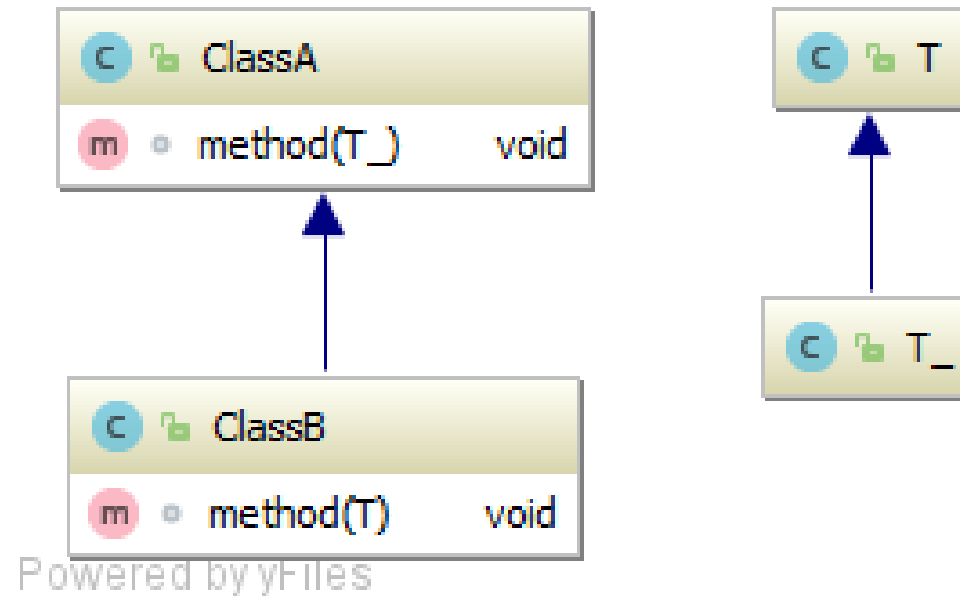
Powered by yFiles

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Kontravarianz (Parameter)

„method“ wird nicht überschrieben
sondern überladen.



IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Bivarianz (Parameter)

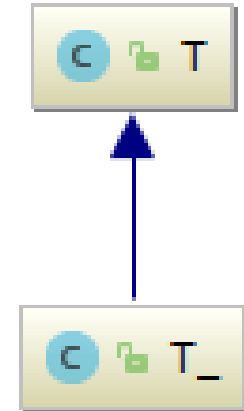
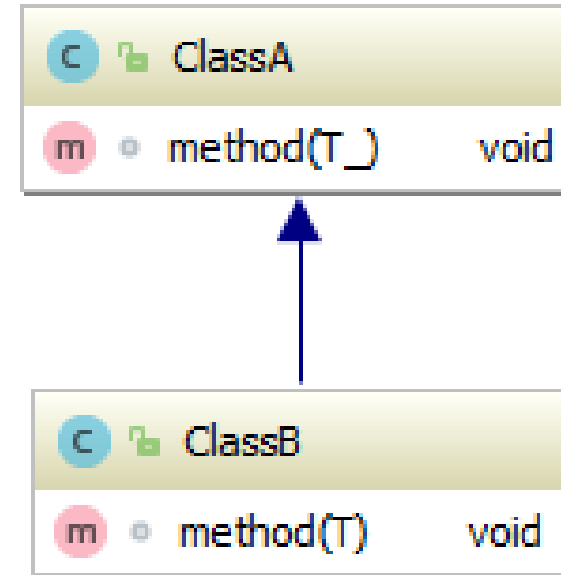
„method“ wird nicht überschrieben
sondern überladen.

Beide „method“ Funktionen würden
jeweils überschrieben werden.

Und invertiert!

Invarianz

Keine Bivarianz bzgl. des
Parameters in Java.

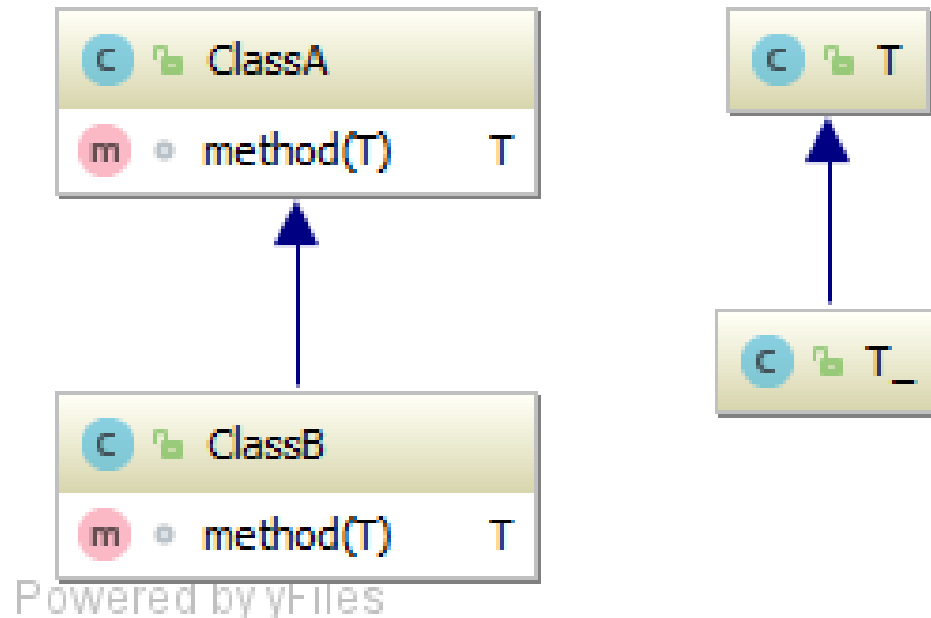


Created by yFiles

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

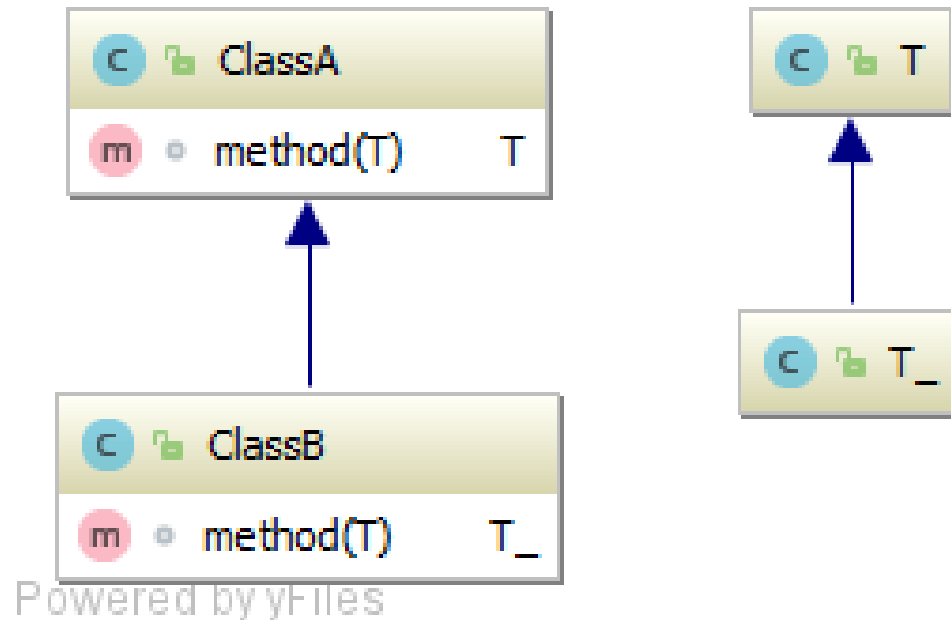
Invarianz (Return Type)



IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Kovarianz (Return Type)



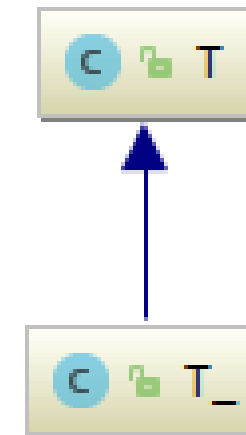
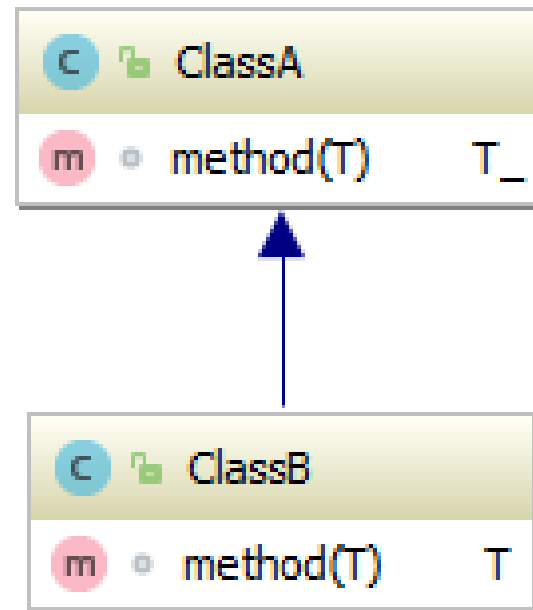
IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Kontravarianz (Return Type)

Compilerfehler:

Keine Kontravarianz bzgl. des
Return Types in Java



IV. Design Patterns

Liskov'sche Substitutionsprinzip:

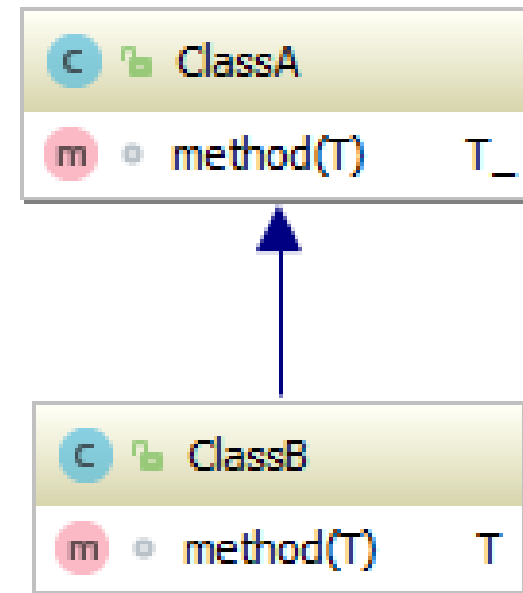
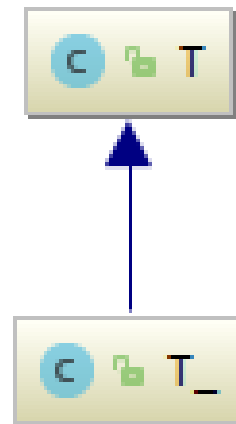
Bivarianz (Return Type)

Compilerfehler:

Keine Bivarianz bzgl. des
Return Types in Java



Keine Kontravarianz bzgl. des
Return Types in Java



Und invertiert

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

varianz ()

- A kann jederzeit durch B ausgeprägt werden, jedoch wird das Verhalten von diesem ,stellvertretenden A' durch die *varianz bestimmt.
- Mit anderen Worten: Die Varianztypen sind Fallunterscheidungen zur Verhaltensdefinition von Eltern-Kind Klassenbeziehung.
- In Java sind folgende Varianzen ,verboten':
 - Bivarianz beim return type
 - Contravarianz beim return type
- Wir sagen: „A und B“ sind hinsichtlich der Methode ,method' Parameter / Return Type (In/Co/Contra/Bi)variant.

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

varianz ()

- In folgenden Fällen wird die Methode der Elternklasse überschrieben:
 - Invarianz bzgl. Return Type und Parameter
 - Kovarianz bzgl. Return Type und Invarianz bzgl. Parameter.
- Allgemein sprechen wir von *varianz:
 - Invarianz, hinsichtlich der Vererbungshierarchie gibt es keine Änderung.
 - Kovarianz, hinsichtlich der Vererbungshierarchie gibt es eine Änderung in Richtung der Spezialisierung.
 - Kontravarianz, hinsichtlich der Vererbungshierarchie gibt es eine Änderung in Richtung der Generalisierung.
 - Bivarianz, hinsichtlich der Vererbungshierarchie gibt es eine Änderung in Richtung der Generalisierung oder Spezialisierung.

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

varianz ()

- Mit T typisierte Elemente E_T (z.B. Arrays) einer Sprache werden als *variant bezeichnet falls:

Kovariant falls: $S \subseteq T \Rightarrow E_S \subseteq E_T$

Kontravariant falls: $S \subseteq T \Rightarrow E_T \subseteq E_S$

Bivariant falls: $S \subseteq T \Rightarrow E_S \subseteq E_T \wedge E_T \subseteq E_S$

Invariant falls: $S \subseteq T \Rightarrow E_S = E_T$

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Wo war eigentlich hier das Liskov'sche Substitutionsprinzip?

$f(x) := \text{,Ist } x \text{ vom Typ ClassA?}'$

IV. Design Patterns

Liskov'sche Substitutionsprinzip:

Generics Revisited

Arrays sind in Java Kovariant:

- ...

Fortsetzung in der nächsten Vorlesung