Objektorientierte Programmierung

Hochschule Bochum
WS 19/20
Dr.-Ing. Darius Malysiak

IV. Design Patterns

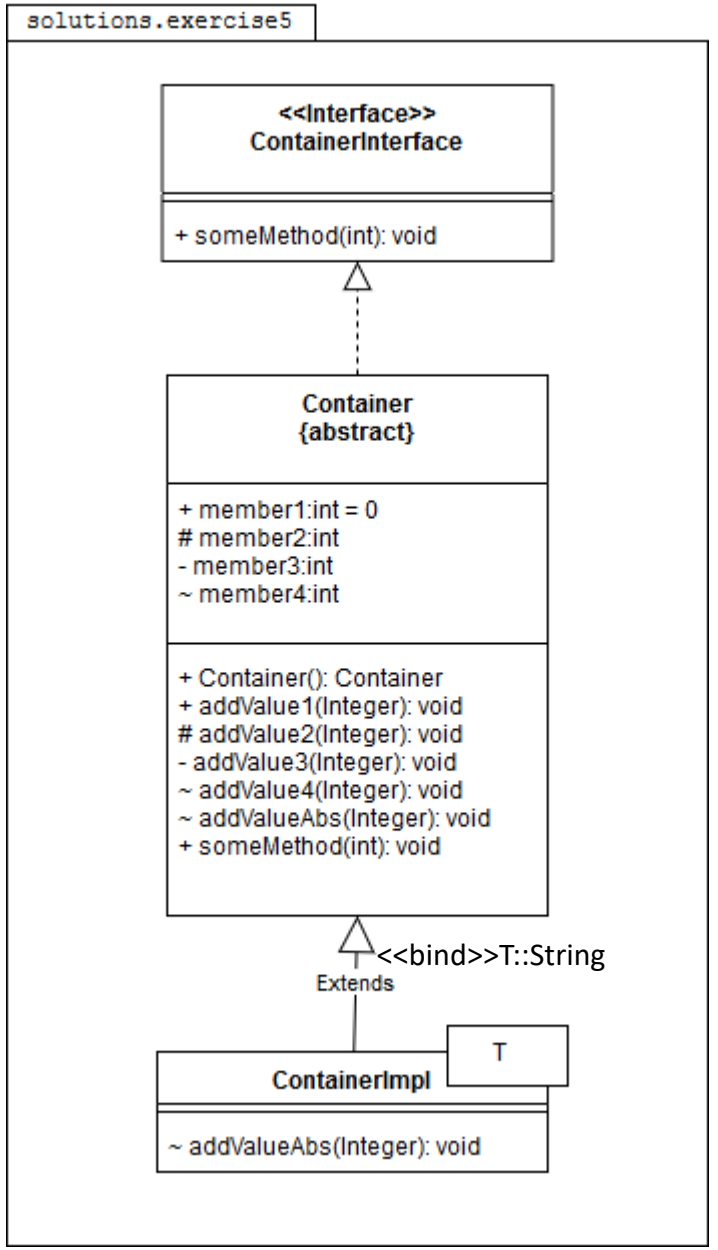
Unified Modeling Language (UML):

- ***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

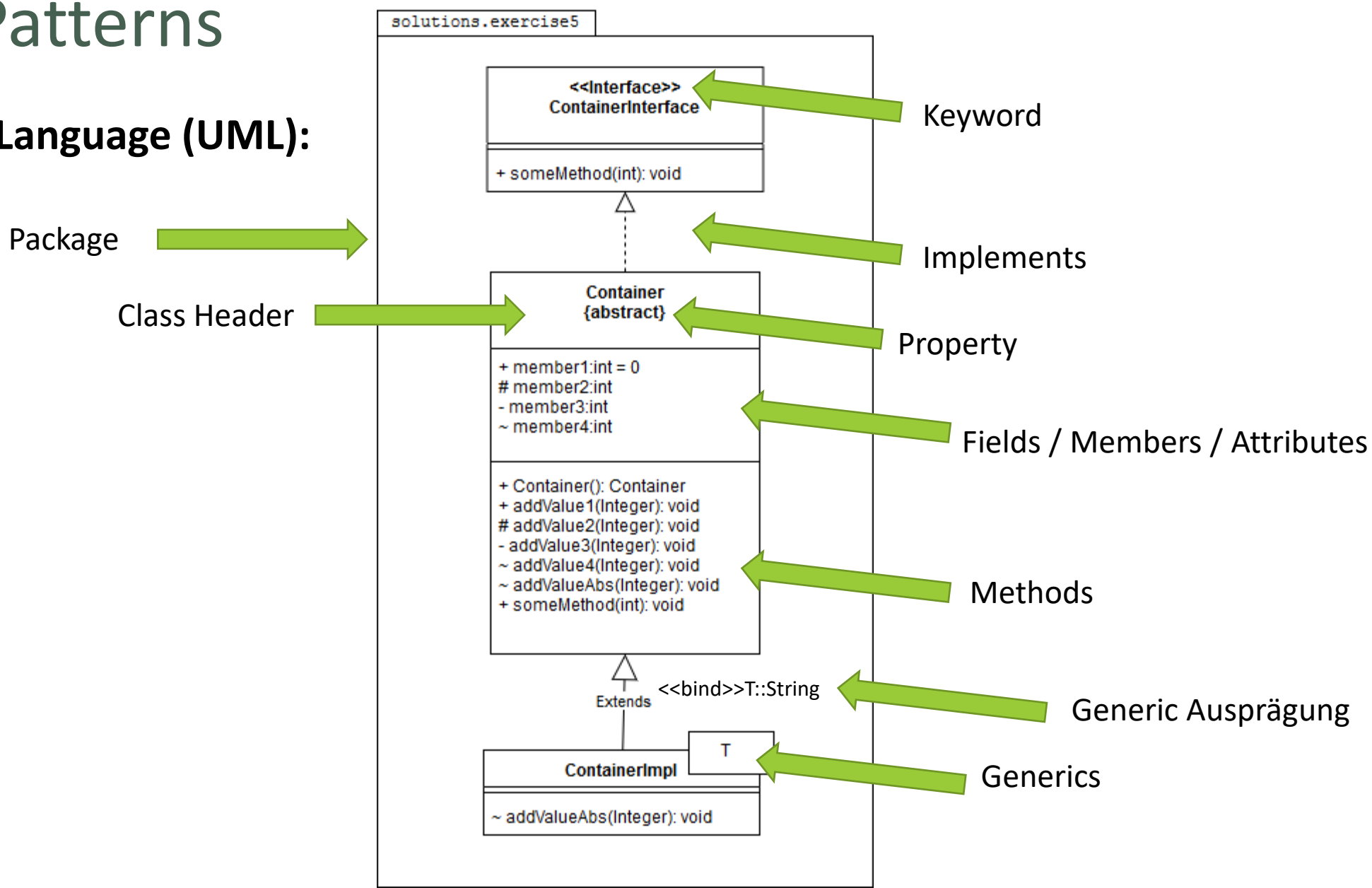
Unified Modeling Language (UML):

```java
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):

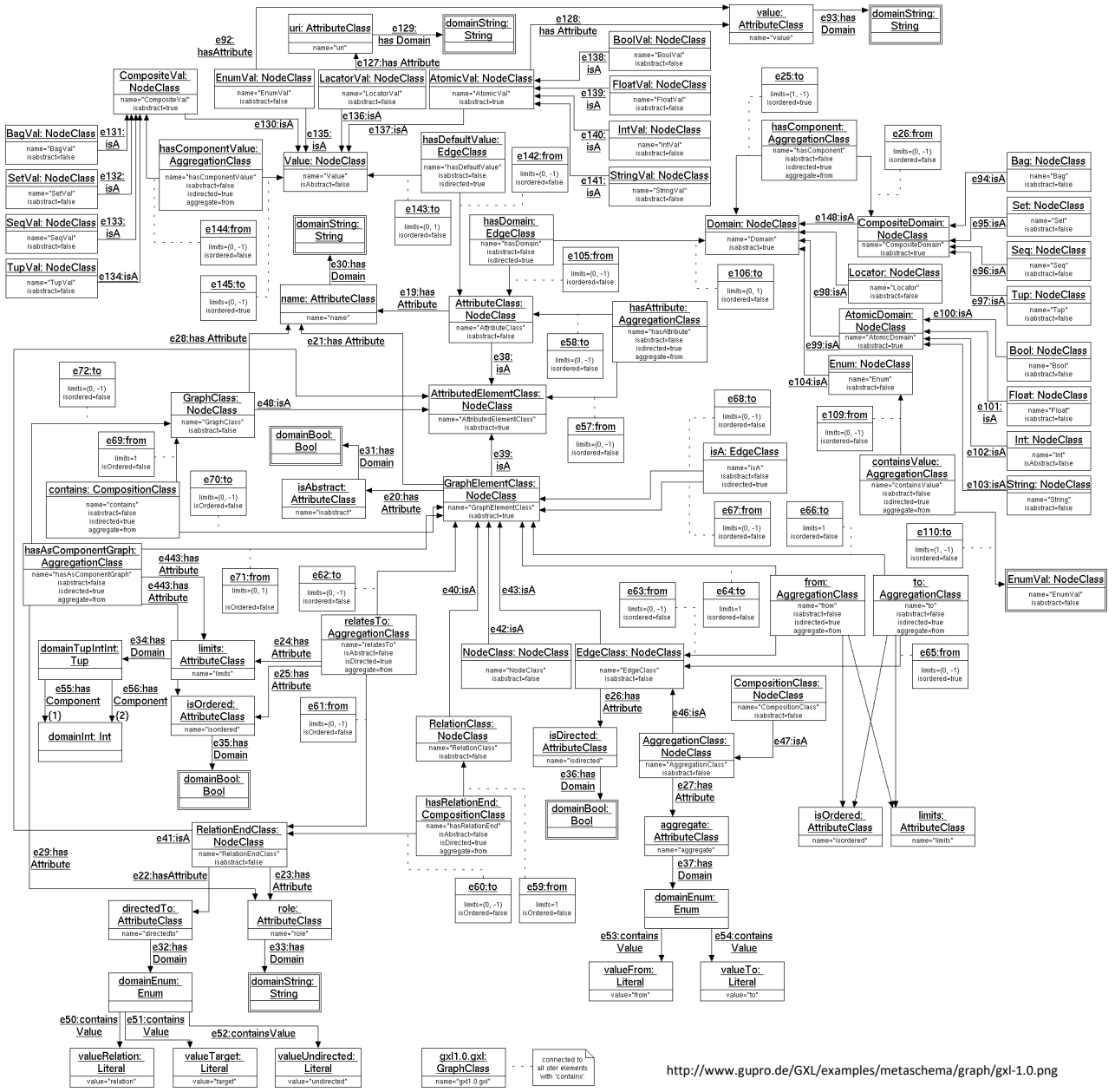
Was ist das Ziel von UML in der Praxis ?

- 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

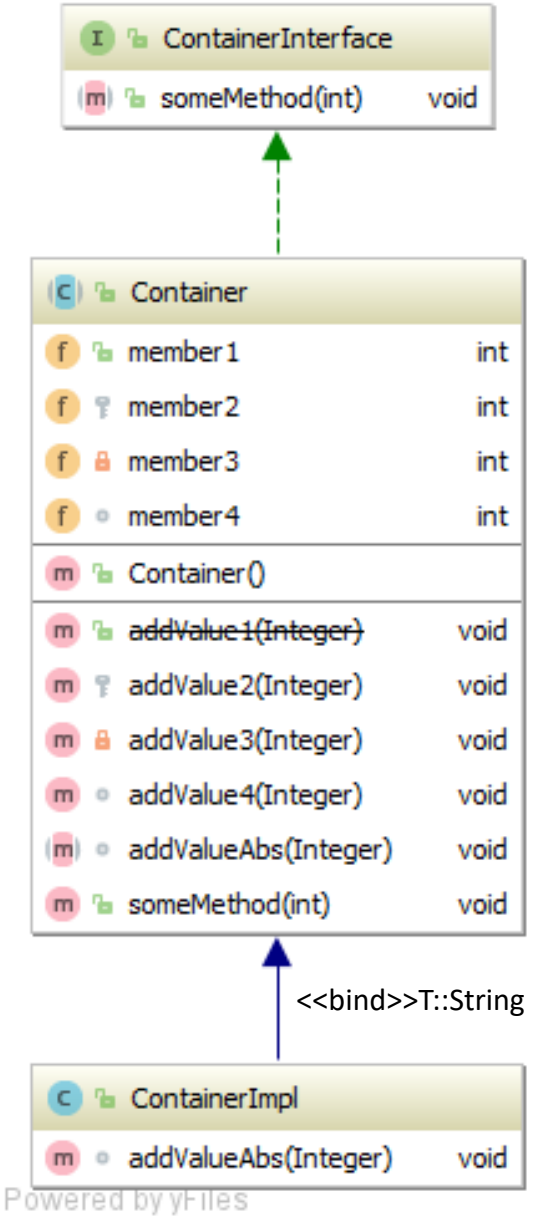


http://www.gupro.de/GXL/examples/metaschema/graph/gxl-1.0.png

IV. Design Patterns

Unified Modeling Language (UML):

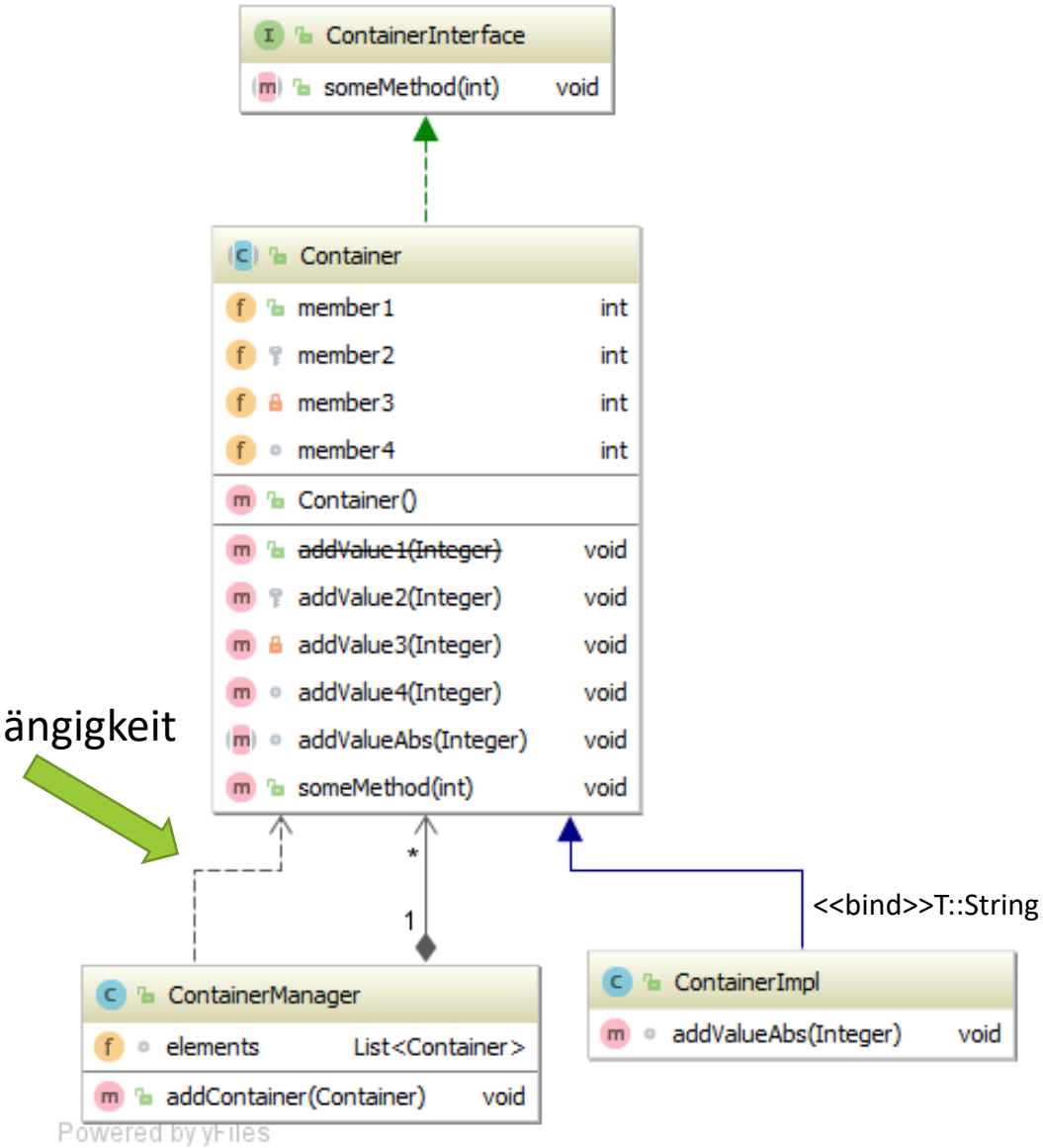
```
public interface ContainerInterface {
    public void someMethod(int value);
}

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):

```
public class ContainerManager {

    List<Container> elements = new ArrayList<>();

    public void addContainer(Container c)
    {
        elements.add(c);
    }
}
```



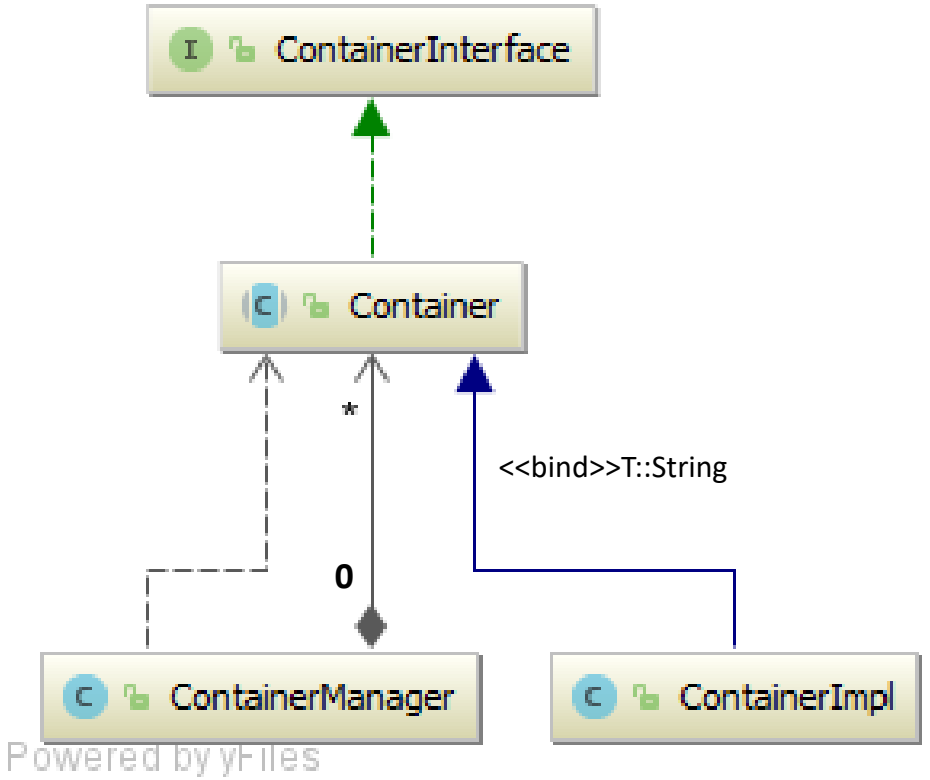
Komposition:

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

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:

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

$$S \subseteq T \land f(T) = \{true\} \land y \in S => f(y) = true$$

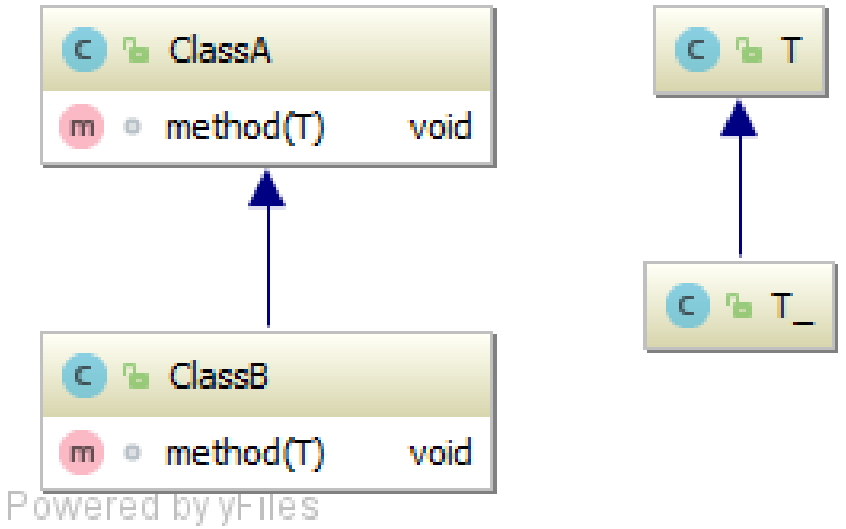
Dieses Prinzip hatte großen Einfluss auf die Entwicklung von Programmiersprachen und implizierte:

- Kovarianz
- Kontravarianz
- Bivarianz
- Invarianz

IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Invarianz (Parameter)

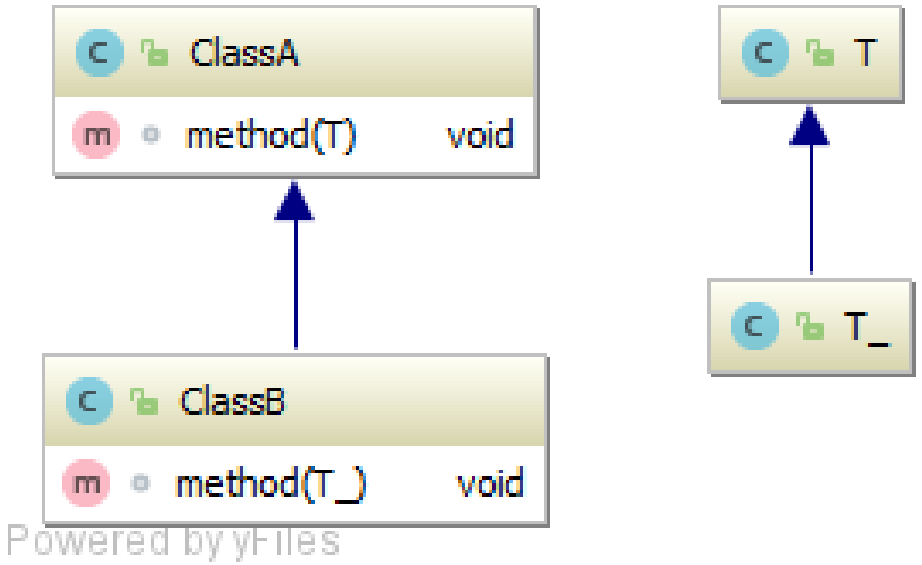


IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Kovarianz (Parameter)

‚method‘ wird nicht überschrieben sondern überladen.

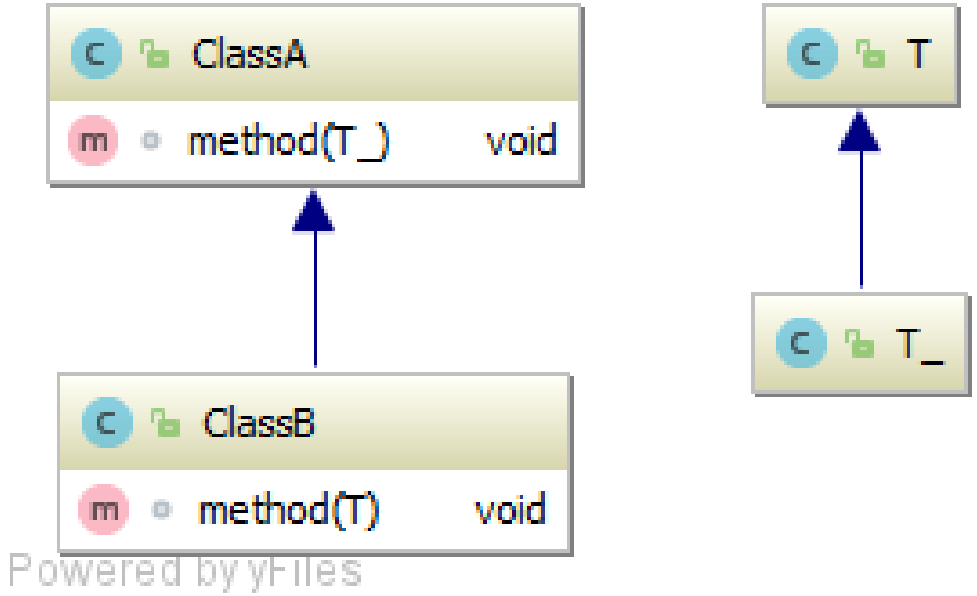


IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Kontravarianz (Parameter)

‚method‘ wird nicht überschrieben sondern überladen.



IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Bivarianz (Parameter)

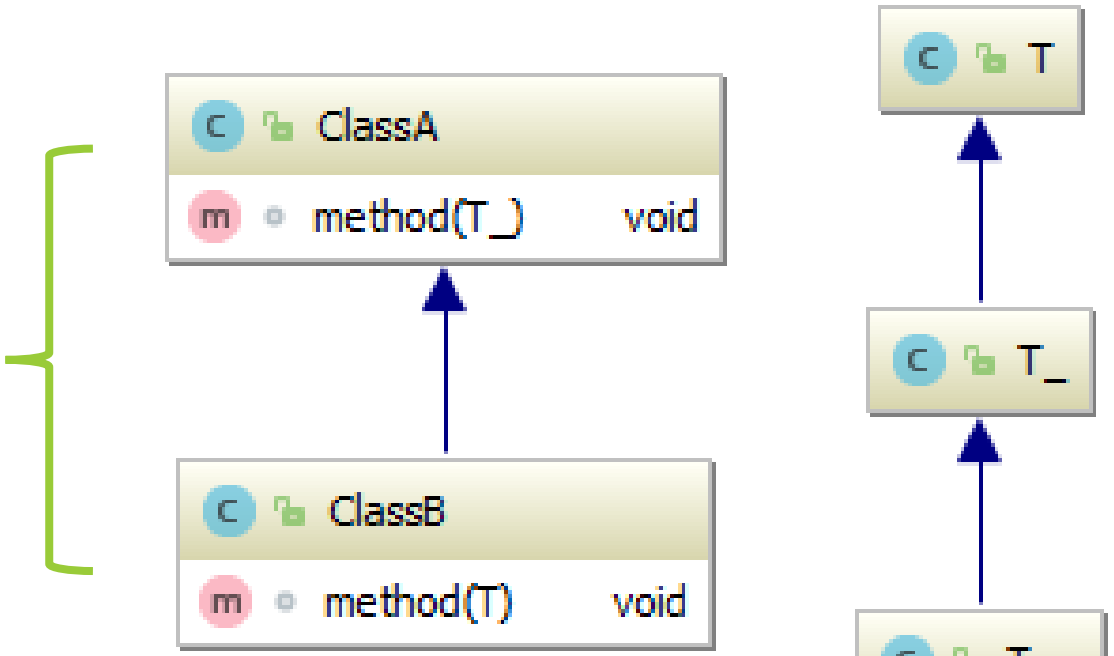
‚method‘ wird nicht überschrieben sondern überladen.

Und invertiert!

Beide ‚method‘ Funktionen würden jeweils überschrieben werden.

Invarianz

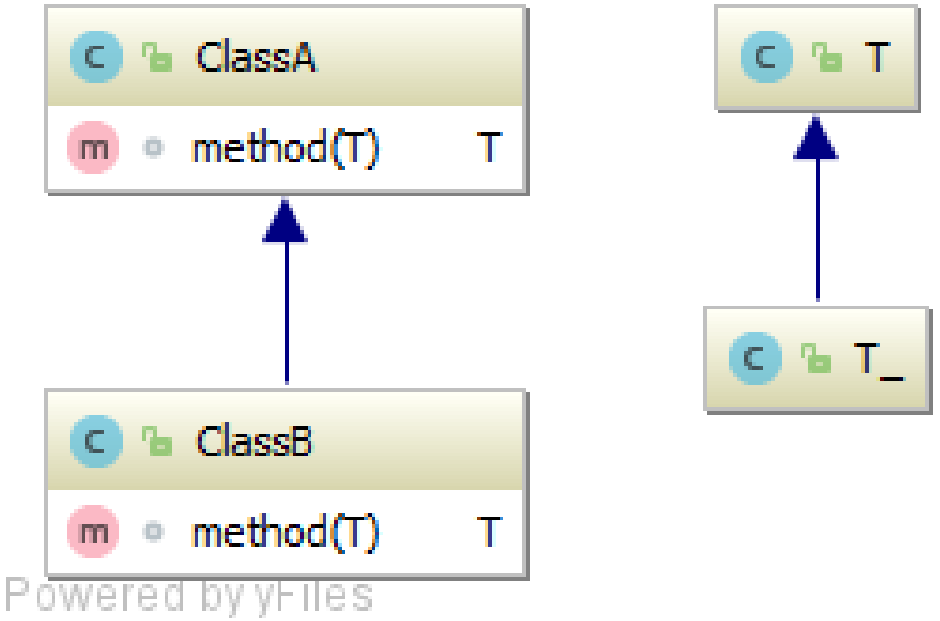
Keine Bivarianz bzgl. des Parameters in Java.



IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

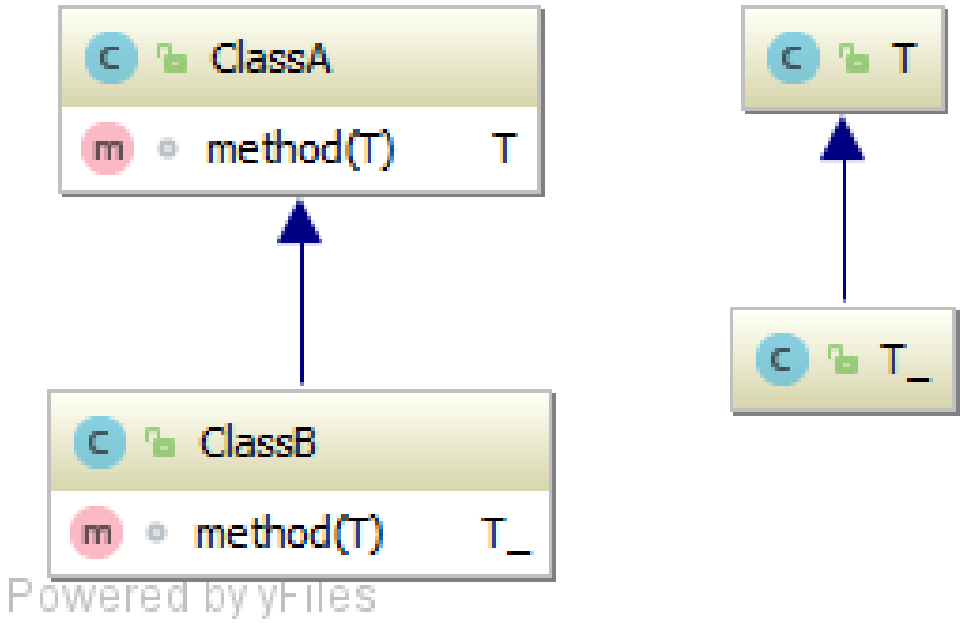
Invarianz (Return Type)



IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Kovarianz (Return Type)



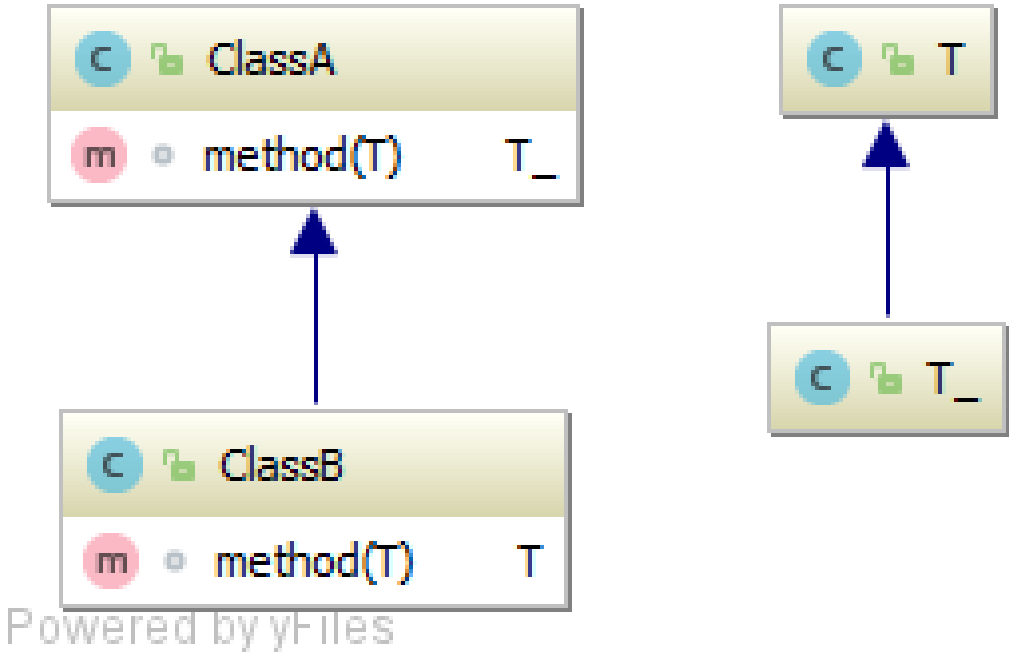
IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Kontravarianz (Return Type)

Compilerfehler: ….

Keine Kontravarianz bzgl. des Return Types in Java



IV. Design Patterns

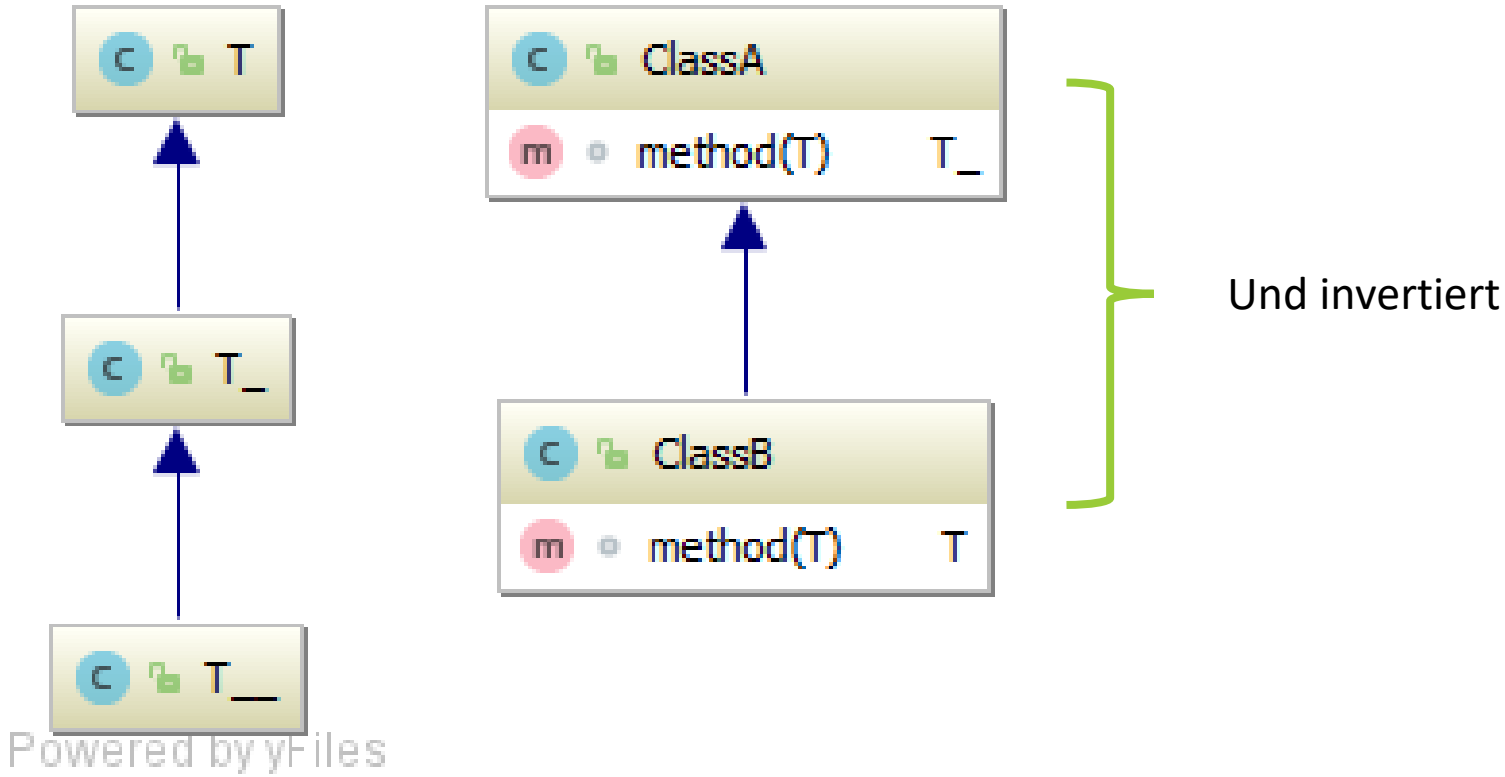
Liskov‘sche Substitutionsprinzip:

Bivarianz (Return Type)

Compilerfehler: ….

Keine Bivarianz bzgl. des Return Types in Java

Keine Kontravarianz bzgl. des Return Types in Java



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:

<u>*varianz (*)</u>

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

$$\text{Kovariant falls: } S \subseteq T => E_S \subseteq E_T$$
$$\text{Kontravariant falls: } S \subseteq T => E_T \subseteq E_S$$
$$\text{Bivariant falls: } S \subseteq T => E_S \subseteq E_T \wedge E_T \subseteq E_T$$
$$\text{Invariant falls: } S \subseteq T => E_S = E_T$$

IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

<u>Wo war eigentlich hier das Liskov‘sche Substitutionsprinzip?</u>

f(x) := ‚Ist x vom Typ ClassA?‘

IV. Design Patterns

Liskov‘sche Substitutionsprinzip:

Generics Revisited

Arrays sind in Java Kovariant:

- …

Fortsetzung in der nächsten Vorlesung