# Objektorientierte Programmierung

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

# IV. Design Patterns

*varianz:

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

## Generics Revisited

Arrays sind in Java Kovariant:

- Sei T_ eine Spezialisierung von T so is T_[] eine Spezialisierung von T[].

```
public static void doSomething(Number[] numbers){}

public static void main(String[] args) {

    Number[] numbers = new Number[2];
    numbers[0] = new Integer(1);
    numbers[1] = new Double(1.2);

    doSomething(numbers);
    doSomething(new Integer[3]);
}
```

Das sollte bekannt sein!

# IV. Design Patterns

*varianz:

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

## Generics Revisited

Daraus folgt, dass eine Referenz auf T[] auch auf eine Instanz von T_[] verweisen kann.

```
Integer[] intArr = {1,2};
Number[] numArr = intArr;
numArr[0] = 1.2;
```

- Kompiliert fehlerfrei da hier keine semantische Prüfung erfolgt!
- Zur Laufzeit:

```
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Double
```

# IV. Design Patterns

*varianz (Generics Revisited):

## **Heap Pollution:**

Def.: Falls eine Referenz auf ein Heap-Objekt vom Typ T auf ein Objekt vom Typ T verweist und T sowie T_ kontravariant oder unabhängig sind, so spricht man von Heap-Pollution.

```
Integer[] intArr = {1,2};
Number[] numArr = intArr;
numArr[0] = 1.2;
```

Würde dies zur Laufzeit funktionieren, so würde Heap-Pollution auftreten.

Heap Pollution führt zu Typunsicherheit hinsichtlich der Heap Objekte!

# IV. Design Patterns

*varianz (Generics Revisited):

## **Refiable Types:**

Def.: Sprachelemente wie z.B. Klassen (ohne Generics) oder Arrays, deren Typinformation zur Laufzeit verfügbar ist werden als ‚refiable types' bezeichnet.

```
Integer[] intArr = {1,2};
Number[] numArr = intArr;
numArr[0] = 1.2;
```

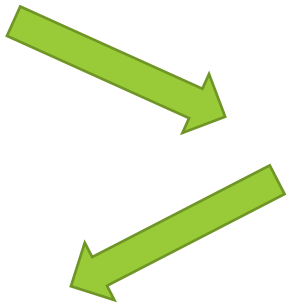
Das Objekt hinter 'intArr' besitzt zur Laufzeit seine Typinformation, aus der Referenz 'numArr'(= intArr) kann jederzeit die Information ausgelesen werden. Die Java Runtime prüft dies bei Zuweisungen!

# IV. Design Patterns

*varianz (Generics Revisited):

**Generics und Refiable Types:**

```
List<Integer> intArr = new ArrayList<>();
intArr.add(1);
intArr.add(2);
List<Number> numArr = intArr;
numArr.add(1.2);
```



Error:(30, 31) java: incompatible types:
java.util.List<java.lang.Integer> cannot be converted to
java.util.List<java.lang.Number>

Objektreferenzen auf Elemente mit ausgeprägten Generics verhalten sich nicht Kovariant!

# IV. Design Patterns

*varianz (Generics Revisited):

## Generics und Refiable Types:

Was wäre wenn der Compiler hier keinen Fehler melden würde?

```
List<Integer> intArr = new ArrayList<>();
intArr.add(1);
intArr.add(2);
List<Number> numArr = intArr;
numArr.add(1.2);
```

**Type-Erasure**

```
List<Object> intArr = new ArrayList<>();
intArr.add(new Integer(1));
intArr.add(new Integer(2));
List<Objekt> numArr = intArr;
numArr.add(new Double(1.2));
```

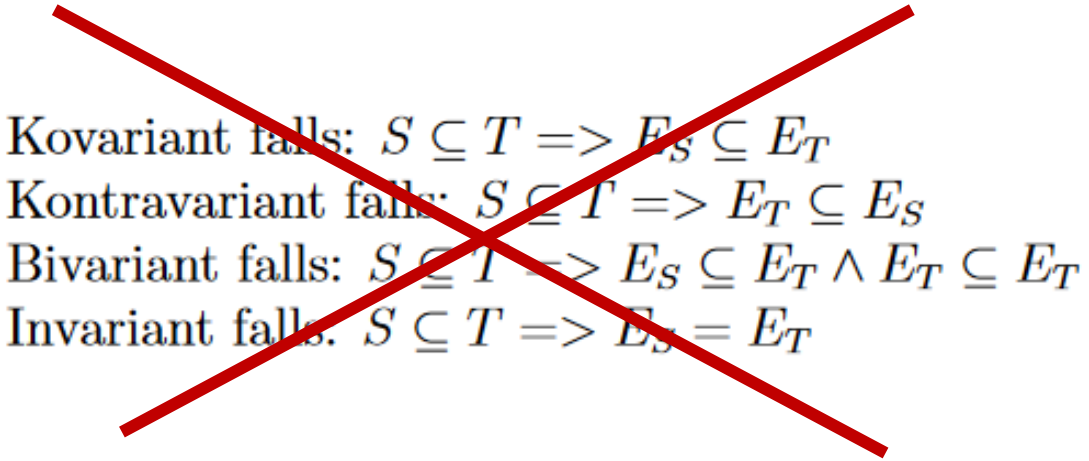
Heap Pollution, da nur Object Referenzen in ‚intArr'.

# IV. Design Patterns

*varianz (Generics Revisited):

**Generics und Refiable Types:**

Generic Klassen sind non-refiable Types!




**Bitte nicht verwechseln**: *varianz bzgl methoden und *varianz bzgl Typen (Liskov'sches Prinzip).

Letzte Vorlesung

Diese Vorlesung → Wichtig bei *varianz bei Methoden mit Generics

# IV. Design Patterns

*varianz (Generics Revisited):

**Generics und Polymorphie (= L-Prinzip mit kovarianten Elementen):**

```
public static void doSomething(List<Number> numbers){}

public static void main(String[] args) {

    List<Integer> intArr = new ArrayList<Integer>();
    intArr.add(1);
    intArr.add(2);
    doSomething(intArr);
}
```

Error:(14, 21) java: incompatible types:
java.util.List<java.lang.Integer> cannot be converted to
java.util.List<java.lang.Number>

**? Lösung hierfür ?**

# IV. Design Patterns

*varianz (Generics Revisited):

**Generics und Polymorphie (= L-Prinzip mit kovarianten Elementen):**

```
public static void doSomething(List<? extends Number> numbers){}
```

```
public static void main(String[] args) {
    List<Integer> intArr = new ArrayList<>();
    intArr.add(1);
    intArr.add(2);
    doSomething(intArr);
}
```

Explizite Angabe von Kovarianz.
Erinnerung: Der Compiler setzt hier Number als Typ.

Die Methode ‚doSomething' kann somit Number-Instanzen aus ‚numbers' aufrufen. Ein Hinzufügen ist jedoch nicht möglich, da sonst zur Laufzeit Heap-Pollution möglich wäre (Folie 7, Number statt Objekt, alles wäre Number)!

Grund für Lese-Restriktion!

# IV. Design Patterns

*varianz (Generics Revisited):

**Generics und Polymorphie (= L-Prinzip mit kovarianten Elementen):**

```
public static void doSomething(List<? super Number> numbers){}

public static void main(String[] args) {

    List<Object> intArr = new ArrayList<Object>();
    intArr.add(1);
    intArr.add(2);
    doSomething(intArr);
}
```

Explizite Angabe von Kontravarianz.
Erinnerung: Der Compiler setzt hier Object als Typ.

Normale Kontravarianz bzgl. der Methode hinsichtlich Parameter!

Die Methode ‚doSomething' kann somit Number-Instanzen in ‚numbers' eintragen. Ein abrufen ist jedoch nicht möglich, da sonst zur Laufzeit Heap-Pollution möglich wäre (Folie 7)!

Grund für Schreib-Restriktion!

# IV. Design Patterns

**Def**.: Eine Architekturschablone / ein Architekturtemplate in der Softwareentwicklung wird als Design Pattern bezeichnet. Hierbei werden diese Muster in 4 Kategorien gruppiert:

- Erzeugungsmuster
- Strukturmuster
- Verhaltensmuster
- Parallelmuster

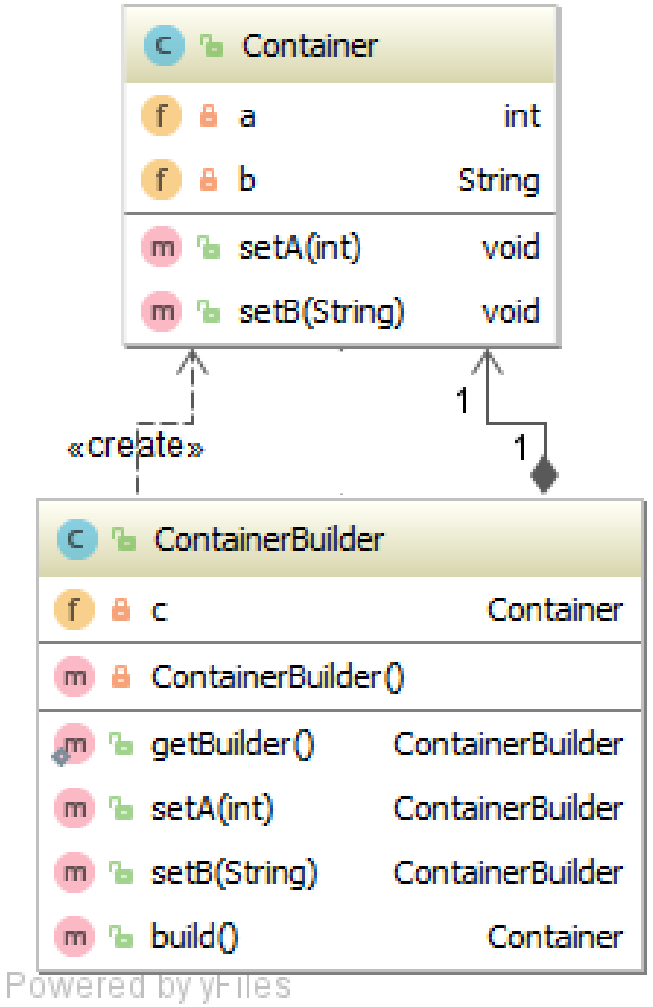
# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

**Ziel**: Ausprägung und Darstellung eines Objekts trennen.

Initialisiertes Objekt — Klasse

```
public static void main(String[] args)
{
Container c =  ContainerBuilder
               .getBuilder()
               .setA(1)
               .setB("Test")
               .build();
}
```



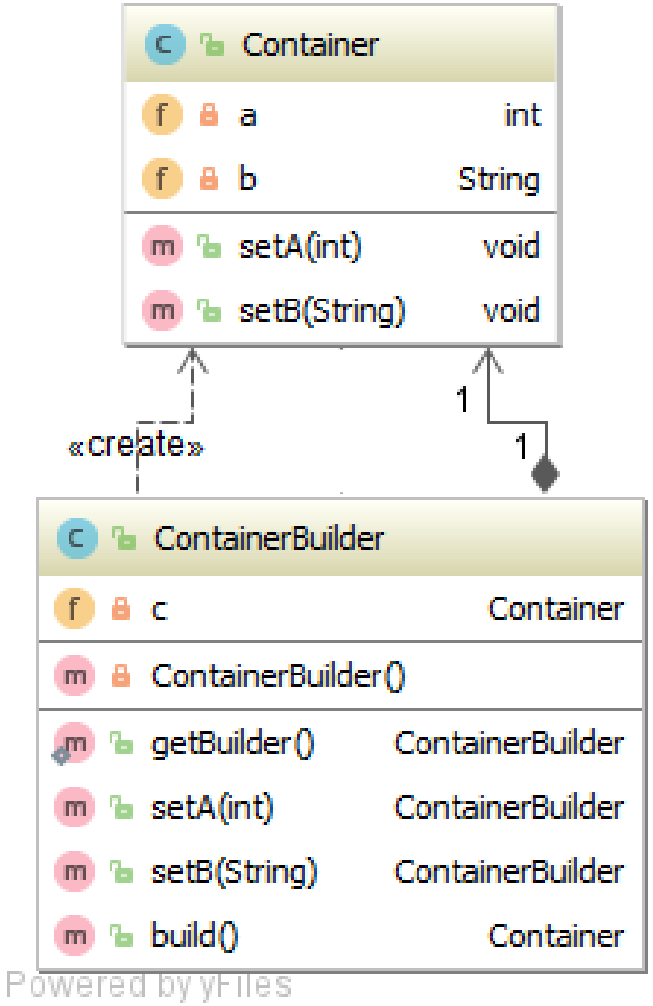
# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

**Ziel**: Ausprägung und Darstellung eines Objekts trennen.

Initialisiertes Objekt — Klasse

```
public static void main(String[] args)
{
Container c =  ContainerBuilder
                .getBuilder()
                .setA(1)
                .setB("Test")
                .build();
}
```




**Live - Übung**

# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

```
public class Container {
    private int a;
    private String b;

    public void setA(int a){this.a=a;}
    public void setB(String b){this.b=b;}
}
```

```
public class ContainerBuilder {
    private Container c = new Container();
    private ContainerBuilder(){}
    public static ContainerBuilder getBuilder()
    {
        ContainerBuilder b = new ContainerBuilder();
        return b;
    }
    public ContainerBuilder setA(int a){c.setA(a);return this;}
    public ContainerBuilder setB(String b){c.setB(b);return this;}
    public Container build() {return c;}
}
```

# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

**Vorteile**:

- Ausprägung muss in keiner Weise vorgegeben werden, Verantwortlichkeit wird an den Builder delegiert.
- Code zur Erzeugung wird implizit gekapselt.
- Erlaubt Kontrolle über die Erzeugung des Objekts z.B. durch Zustandsautomaten im Builder.

**Nachteile**:

- Erfordert eine separater Builderklasse.
- Erfordert die Erzeugung eines Builder Objekts für jedes zu konstruierende Objekt.
- Ohne Zustandsautomat im Builder gibt es keine Garantie für eine vollständige Initialisierung des Objekts bzgl. seiner Felder.
- (Probleme mit Dependency Injection) ← Später mehr!

# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

```
public class Entity {
    private int intValue1;

    @Builder
    public void setIntValue1(int val)
    {
        intValue1 = val;
    }
}
```

**Erinnerung Vorlesung 4**

Test ☺ (später mehr)

Builder Pattern

```
@Test
public void testCreation() {
    Entity e =
      EntityBuilder.get().setIntValue1(1).build();

    assertTrue( e.getIntValue1() == 1 );
}
```

# IV. Design Patterns

**Builder Pattern:** Erzeugungsmuster

**Lombok**

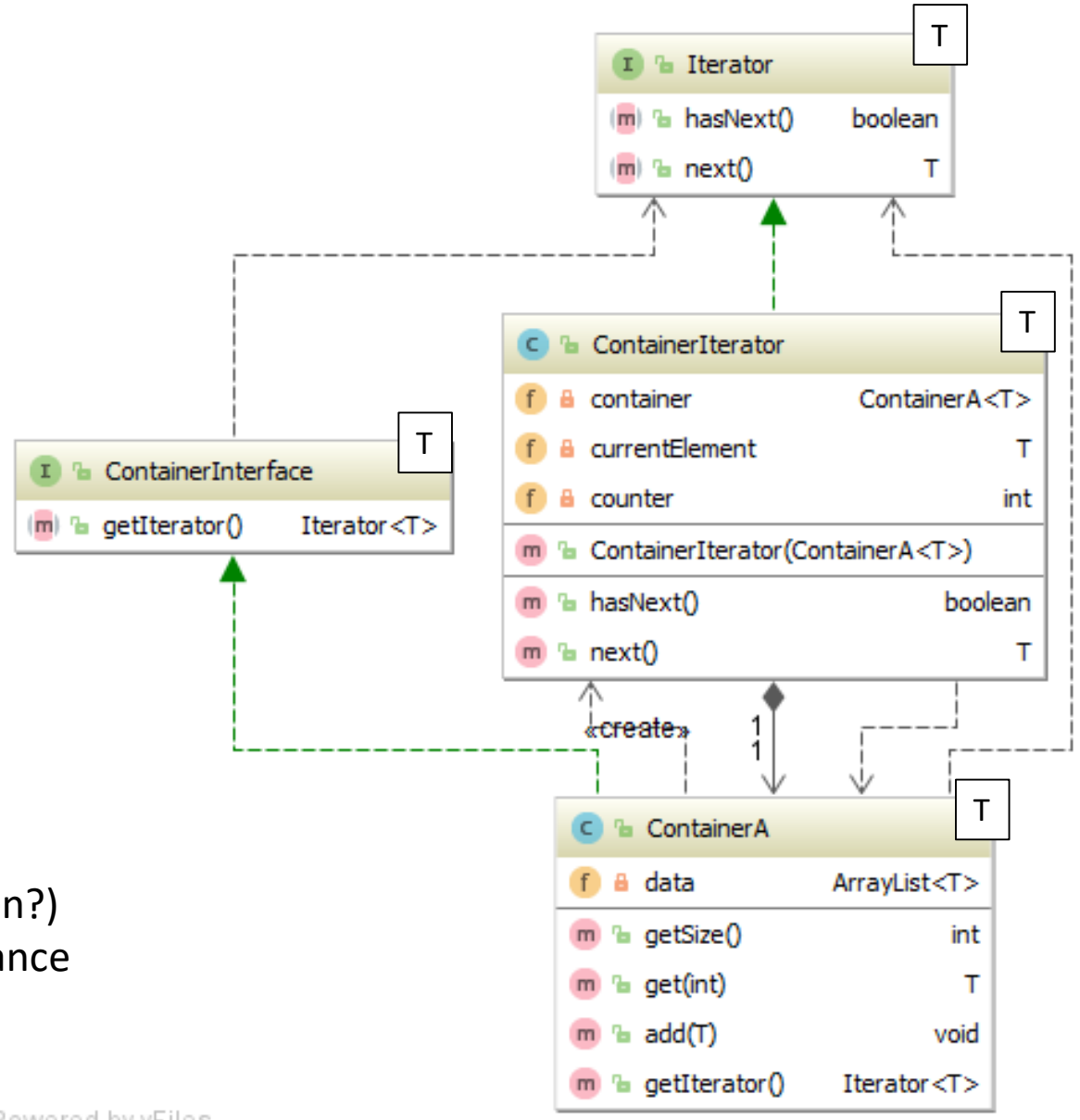
```
@Builder
public class Container {
    private int a;
    private String b;

    public void setA(int a){this.a=a;}
    public void setB(String b){this.b=b;}
}
```

# IV. Design Patterns

**Iterator Pattern:** Verhaltensmuster

**Ziel**: Generisches Traversieren eines Containers




**Vorteile**:

- Container können über identische Logik traversiert werden.
- Container müssen ihre Funktionsweise nicht offenlegen.

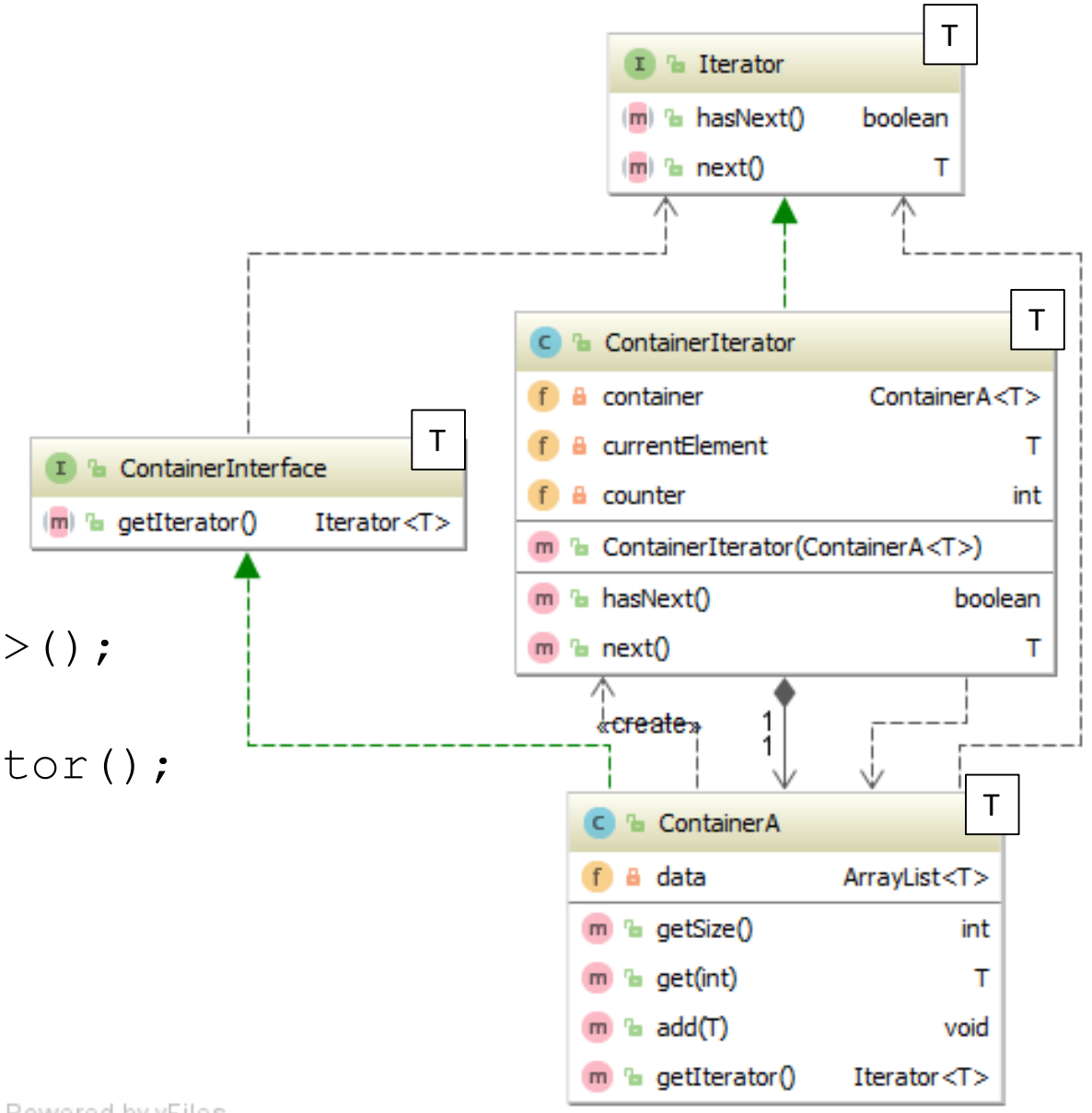
**Nachteile**:

- Implementierung für jede Containerklasse nötig (Annotationen?)
- Häufig ist ein Trade-off zwischen Bequemlichkeit und Peformance nötig.

# IV. Design Patterns

**Iterator Pattern:** Verhaltensmuster

**Ziel**: Generisches Traversieren eines Containers




```
ContainerA<Integer> c2 = new ContainerA<>();
c2.add(2);c2.add(3);c2.add(4);
Iterator<Integer> iterator = c2.getIterator();
while(iterator.hasNext())
{
    Integer next = iterator.next();
    System.out.println(next);
}
```