



TECHNISCHE  
UNIVERSITÄT  
WIEN  
Vienna | Austria



Institut für  
Computertechnik  
Institute of  
Computer Technology

# UML Diagramme

Thomas Rathfux  
Ralph Hoch

# Überblick

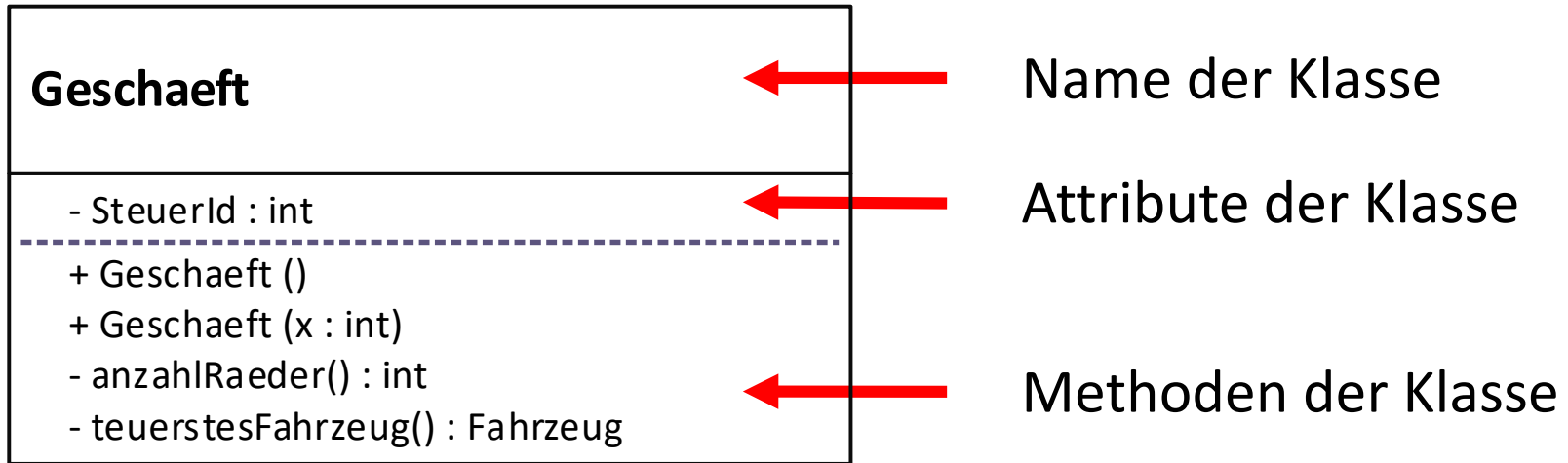
- Klassendiagramme
  - Klassen
  - Abstrakte Klassen
  - Spezialisierung von Klassen
  - Assoziationen
  - Interfaces
- Klassendiagramme - Tools
- Sequenzdiagramme



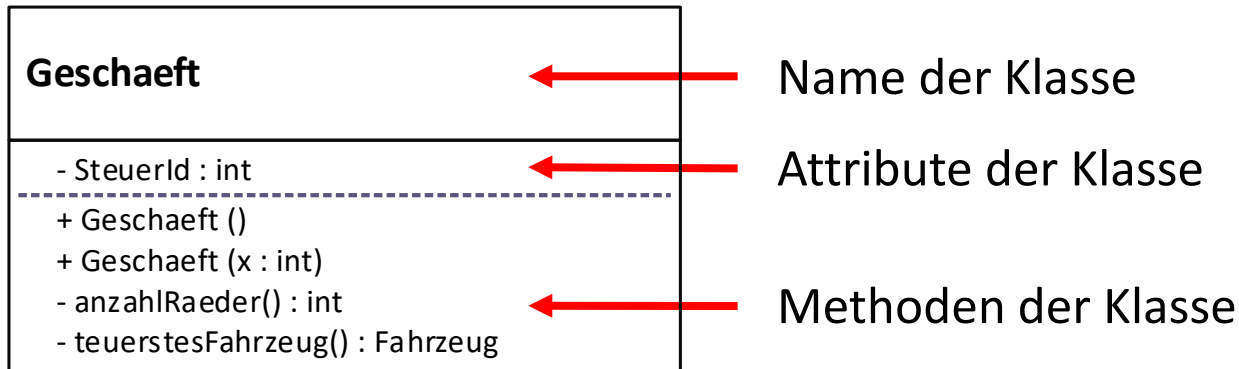


# Klassendiagramme

# Klassen in UML Diagrammen



# Klassen - Attribute



## Attribute

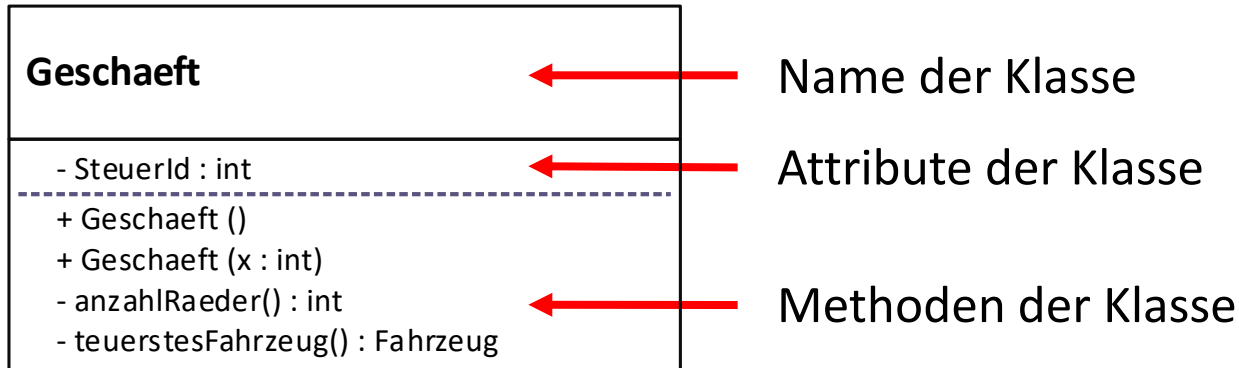
[ Sichtbarkeit ] name [ : Typ ] [ Multiplizität ] [= Vorgabewert]

- SteuerId : int

$$\text{Sichbarkeit} = \begin{cases} + \dots public \\ \# \dots protected \\ - \dots private \end{cases}$$



# Klassen - Methoden



## Methoden

[ Sichtbarkeit ] name [( {Parameter} ) ] [: Rückgabetyp ]

- anzahlRaeder() : int

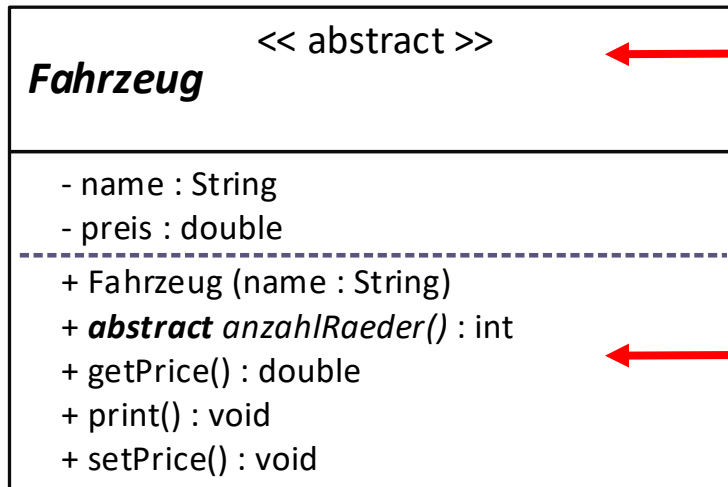
- teuerstesFahrzeug() : Fahrzeug

# Klassen

```
public class Geschaeft {  
    // Attribute  
    private int SteuerId = 0;  
  
    public Geschaeft(int x) {  
        // Methodenrumpf  
    }  
    public Geschaeft() {  
        // Methodenrumpf  
    }  
    private int anzahlRaeder() {  
        // Methodenrumpf  
    }  
    private Fahrzeug teuerstesFahrzeug() {  
        // Methodenrumpf  
    }  
}
```



# Abstrakte Klassen



Schlüsselwort „abstract“

Jede Klasse die mindestens eine abstrakte Methode beinhaltet muss *abstract* sein.

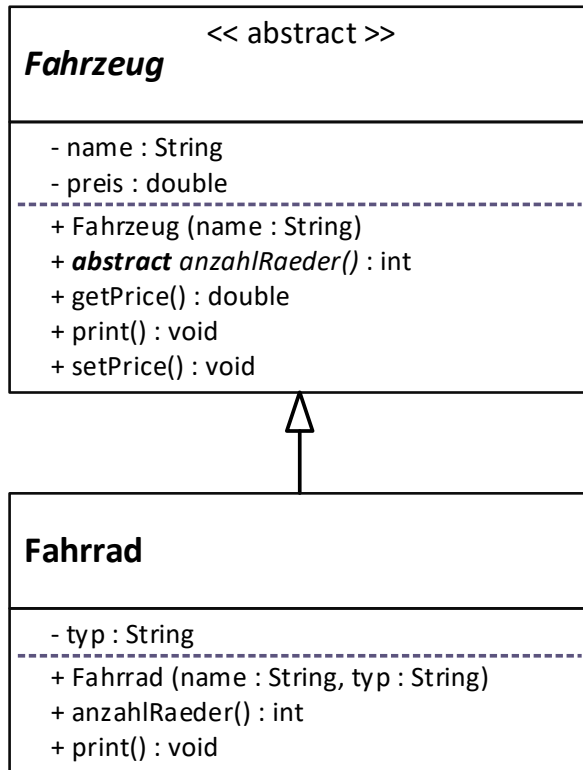
**Hinweis:** Abstrakte Klassen werden im UML Standard als Klasse, mit einem kursiven Klassennamen definiert. Aufgrund der einfacheren Erkennbarkeit wird aber auch oft ein abstract-Stereotyp <<*abstract*>> angegeben. Abstrakte Methoden werden kursiv dargestellt, auch hier wird oft zusätzlich noch der *abstract*-Stereotyp angegeben.



# Abstrakte Klassen

```
public abstract class Fahrzeug {  
  
    private String name = null;  
    private double preis = 0;  
  
    public Fahrzeug(String name) { // Methodenrumpf }  
  
    public void setPreis(double preis) { // Methodenrumpf }  
  
    public double getPreis() { // Methodenrumpf }  
  
    public void print() { // Methodenrumpf }  
  
    public abstract int anzahlRaeder();  
}
```

# Spezialisierung von Klassen



```

public abstract class Fahrzeug {
    private String name = null;
    private double preis = 0;

    public Fahrzeug(String name) {
        // Methodenrumpf
    }

    // ... Methoden der Klasse ...
    public abstract int anzahlRaeder();
}
  
```

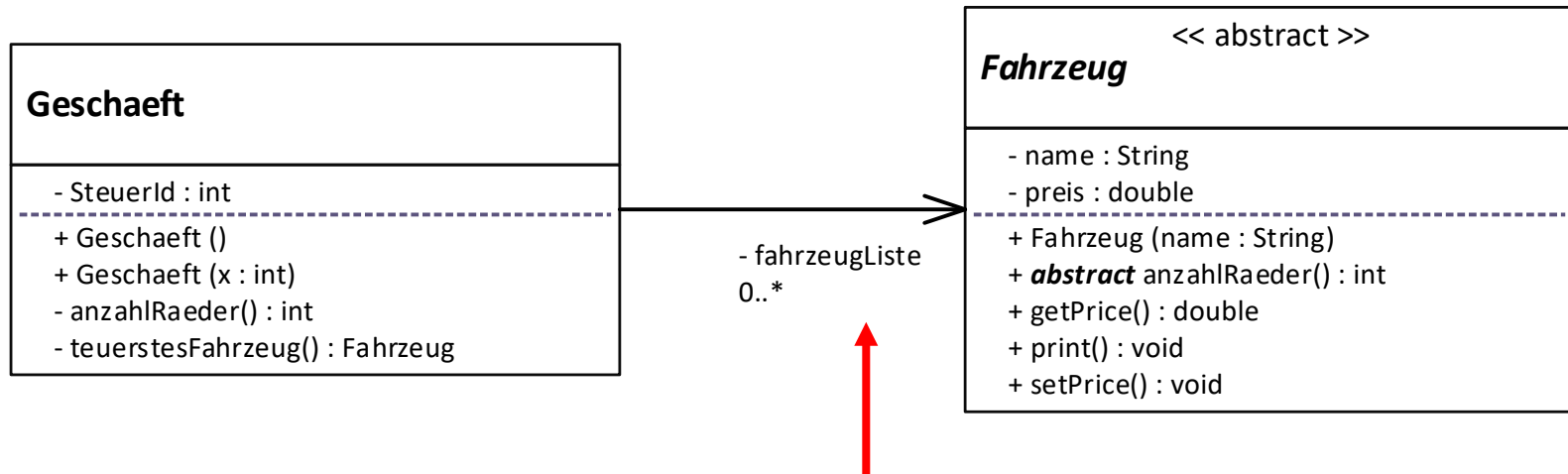
```

public class Fahrrad extends Fahrzeug {
    private String typ = null;

    public Fahrrad(String name, String typ) {
        // Methodenrumpf
    }

    // ... Methoden der Klasse ...
    public int anzahlRaeder() {
        // Methodenrumpf
    }
}
  
```

# Assoziationen



## Gerichtete Assoziation:

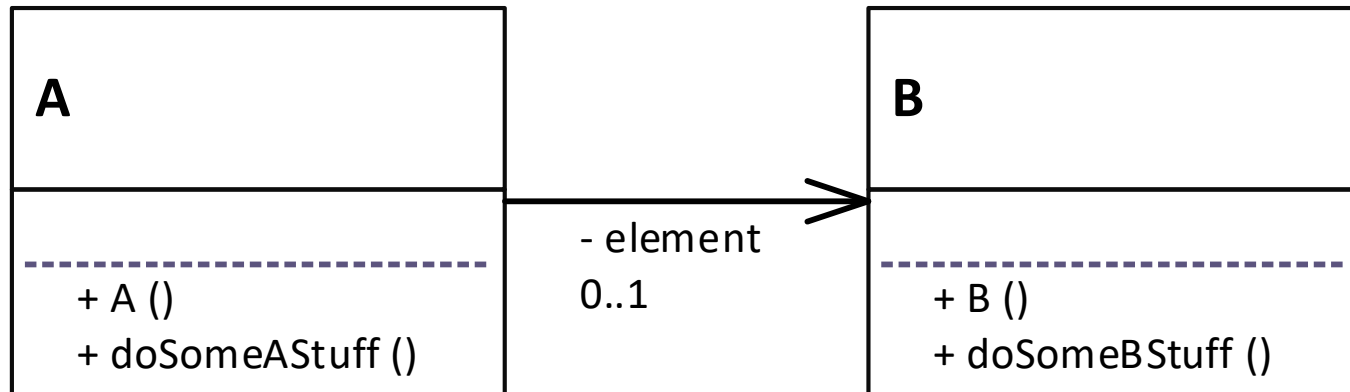
*Erlaubte Navigation:* Das Modell erlaubt die Navigation über das Assoziationsende.

Eigenschaften: Sichtbarkeit, Rollenname und Multiplizität

$$\text{Multiplizität} = \begin{cases} \text{untereGrenze} \dots \text{obereGrenze} \\ 0..1 \\ 0..* \end{cases}$$

$$\text{Sichbarkeit} = \begin{cases} + \dots \text{public} \\ \# \dots \text{protected} \\ - \dots \text{private} \end{cases}$$

# Assoziationen



```
public class A {
    private B element = null;
    public A() { // Konstruktor }

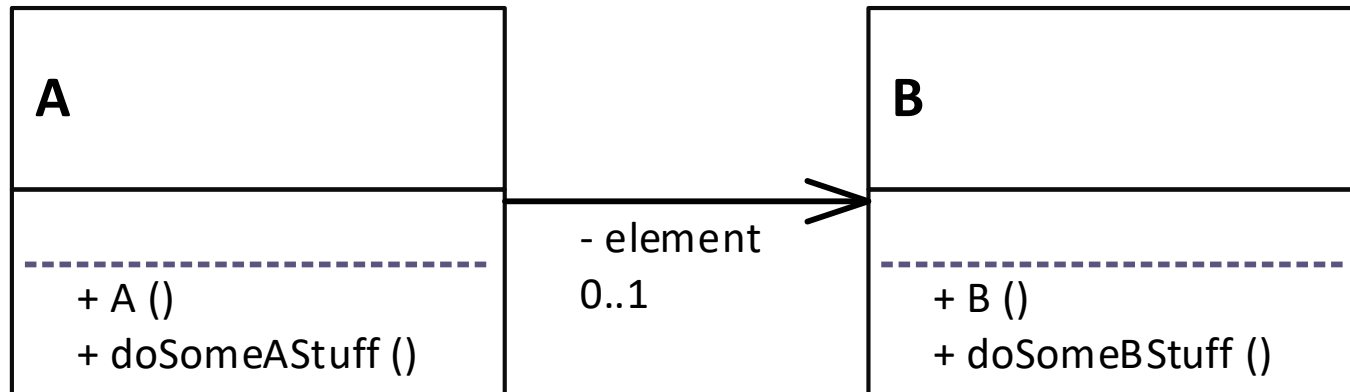
    // ... Methoden der Klasse ...
}
```

```
public class B {

    public B() { // Konstruktor }

    // ... Methoden der Klasse ...
}
```

# Assoziationen – Navigierbarkeit



```

public class A {
    private B element = null;
    public A() { // Konstruktor }

    public void doSomeAStuff() {
        element.doSomeBStuff();
    }

    // ... Methoden der Klasse ...
}
  
```

Navigierbarkeit

```

public class B {

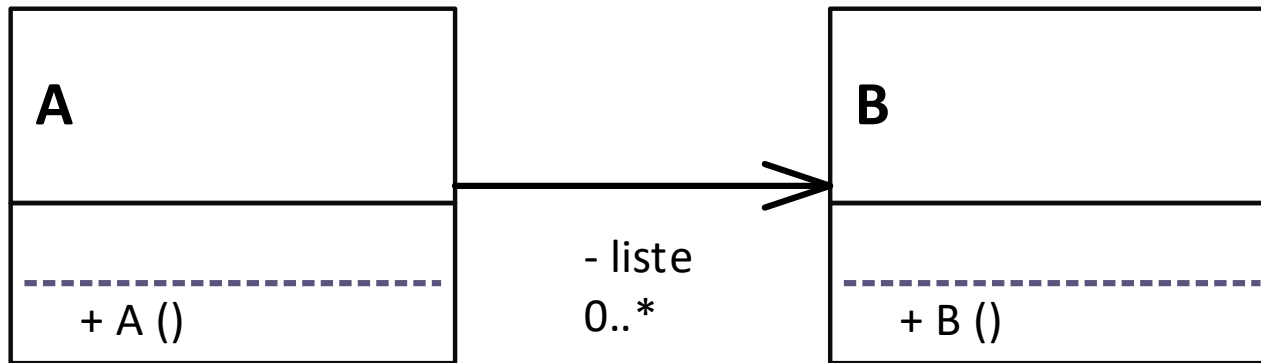
    public B() { // Konstruktor }

    public void doSomeBStuff() {

    }

    // ... Methoden der Klasse ...
}
  
```

# Assoziationen



```
public class A {
    private Collection<B> liste = null;

    public A() { // Konstruktor }

    // ... Methoden der Klasse ...
}
```

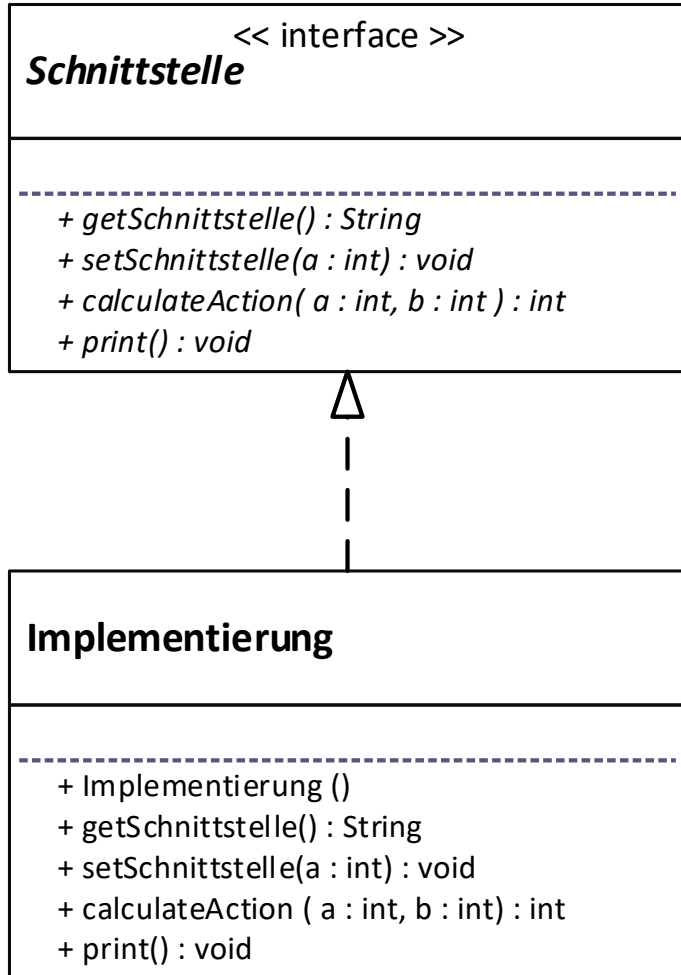
Assoziationen mit einer Multiplizität größer als 1 werden in Java üblicherweise durch *Collections* oder abgeleitete Typen abgebildet.

```
public class B {

    public B() { // Konstruktor }

    // ... Methoden der Klasse ...
}
```

# Interfaces

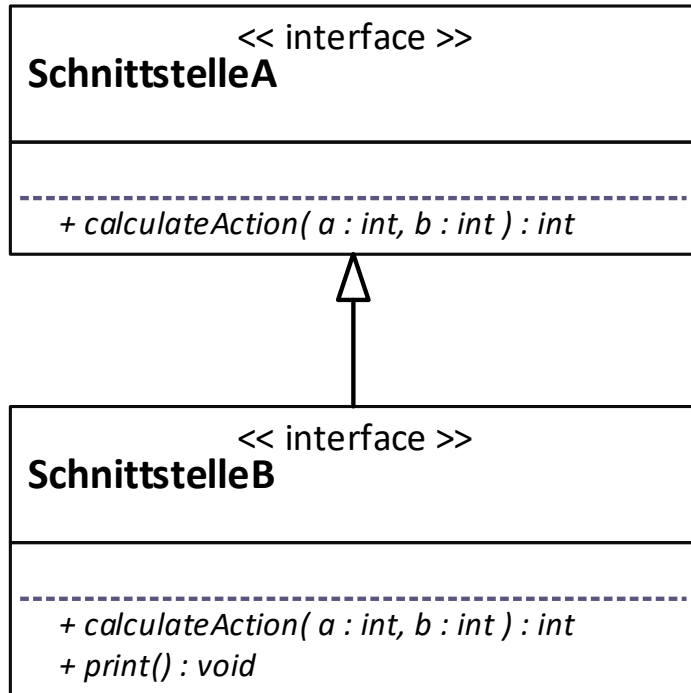


```
public class Implementierung
    implements Schnittstelle {

    public Implementierung() {
        // Konstruktor
    }
    public String getSchnittstelle() {
        // ...
    }
    public void setSchnittstelle(int a) {
        // ...
    }
    public void calculateAction(int a, int b) {
        // ...
    }
    public void print() throws Exception {
        // geworfene Exception wird aus der
        // Methodendokumentation entnommen.
    }
}
```



# Interfaces



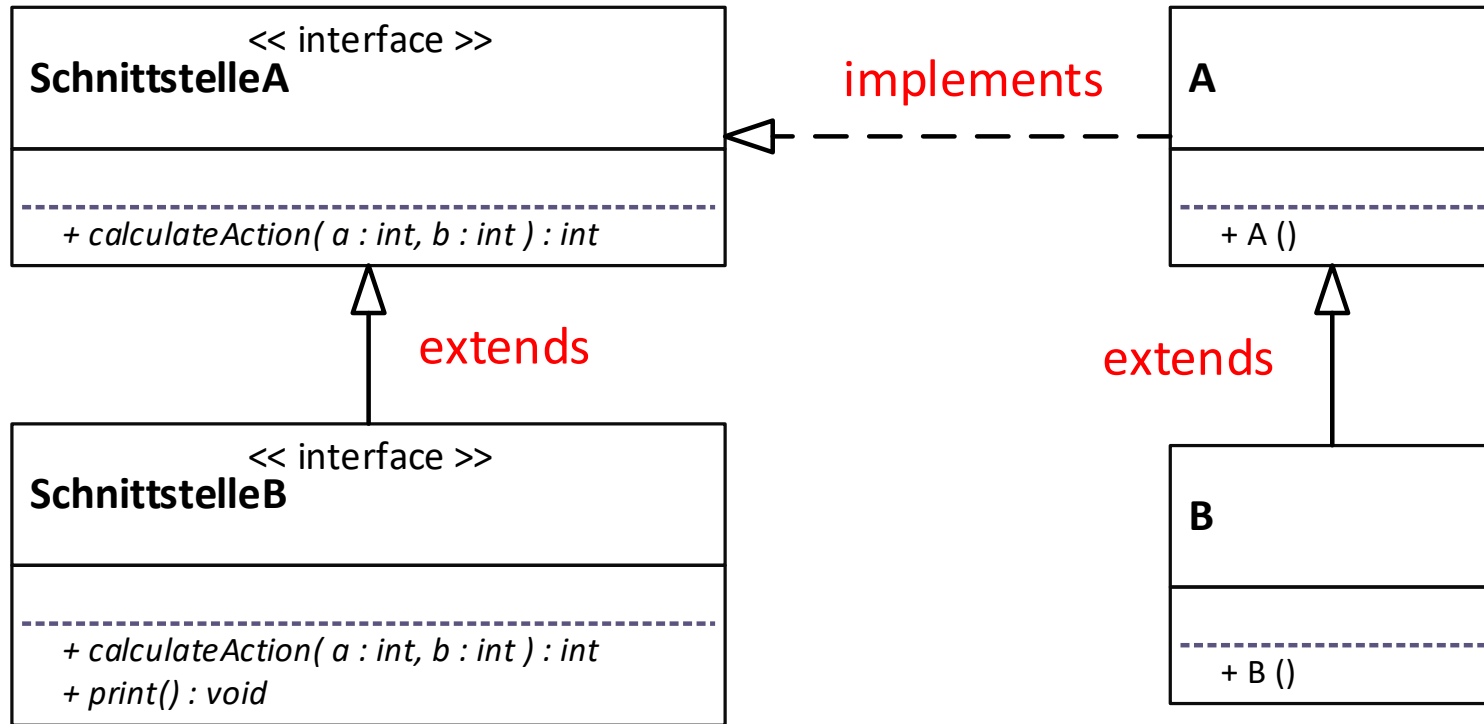
```
public interface SchnittstelleB extends
    SchnittstelleA {

    public void calculateAction(int a, int b);
    public void print() throws Exception;
}
```





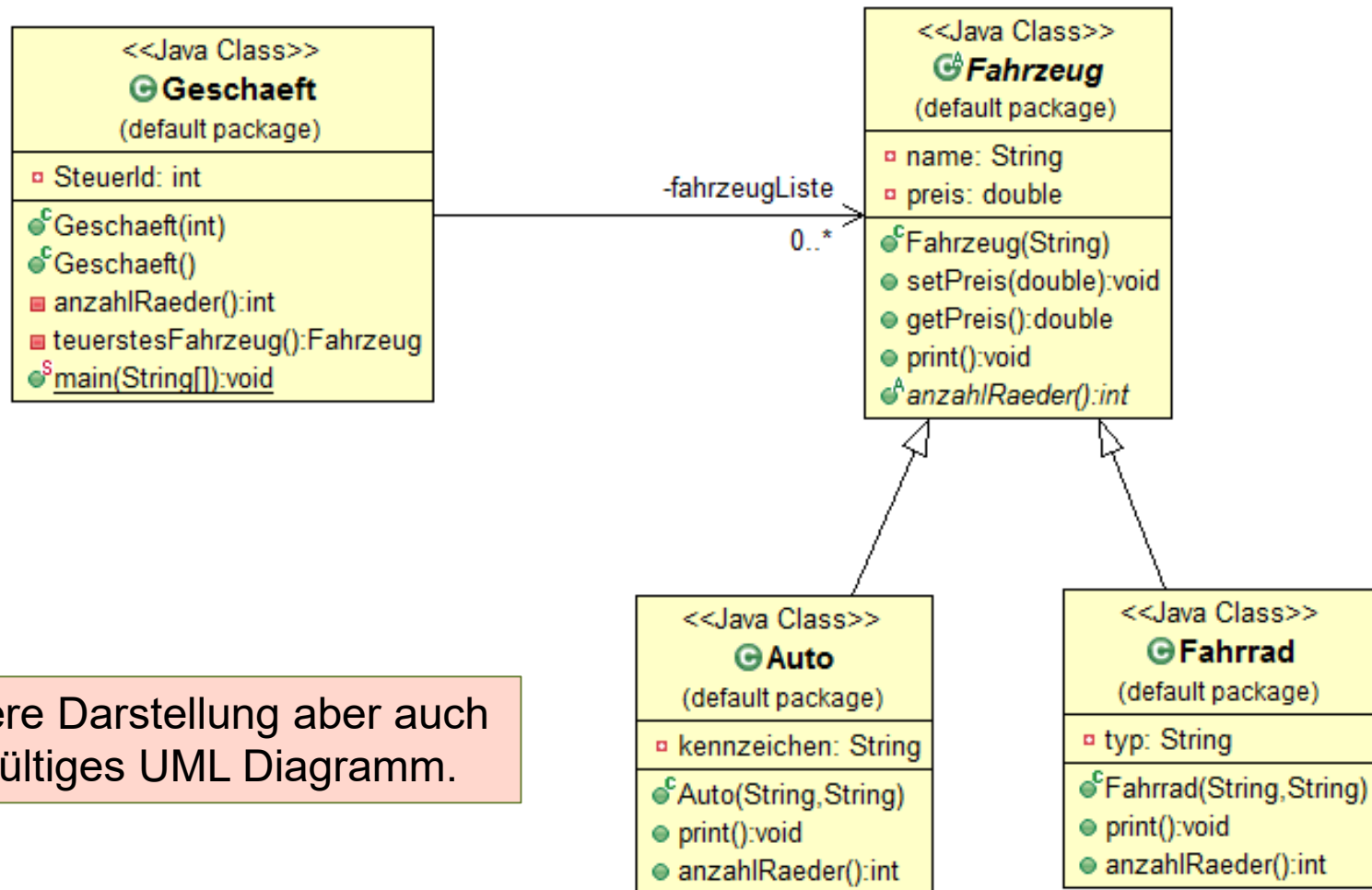
# Spezialisierung Interfaces / Klassen



- Java **Schlüsselwörter** bei der Spezialisierung
- **Kontrollfrage:** Welche Methoden bieten Klasse A und Klasse B ?


# Klassendiagramme - Tools




# ObjectAid – Eclipse Plugin

















Andere Darstellung aber auch ein gültiges UML Diagramm.

# ObjectAid – Eclipse Plugin

<<Java Interface>>	
<b>I</b>	<b>InterfaceDefinition</b>
(default package)	
	interfaceMethod():void

<<Java Class>>	
	<b>AbstractClass</b>
(default package)	
	AbstractClass()
	abstractMethod():String

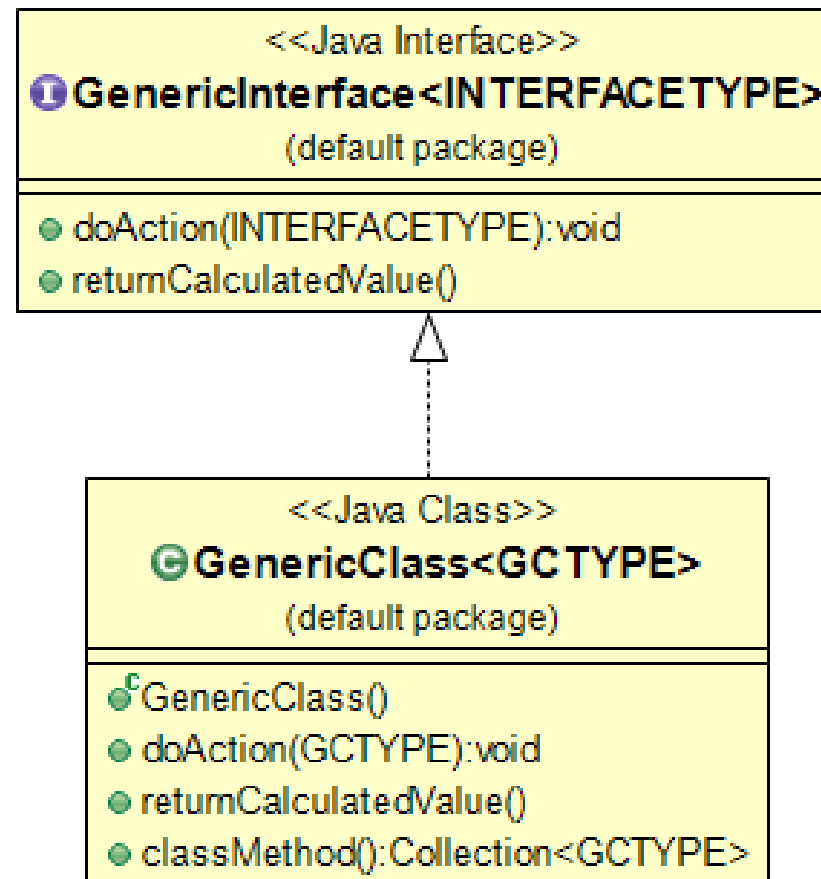
<<Java Enumeration>>	
<b>E</b>	<b>EnumClass</b>
(default package)	
	Value1: EnumClass
	Value2: EnumClass
	Value3: EnumClass
	EnumClass()
	valueOfMethod(EnumClass):String

<<Java Class>>	
	<b>ObjectClass</b>
(default package)	
	publicAttribute: String
	protectedAttribute: Integer
	privateAttribute: Float
	ObjectClass()
	publicMethod():String
	publicStaticMethod():String
	protectedMethod():Integer
	privateMethod():Float

# ObjectAid – Eclipse Plugin

- Sichtbarkeit
  - Rotes Rechteck = Privat
  - Gelbe Raute = Protected
  - Grüner Kreis = Public
- Symbole in Diagram:
  - A .. Abstract (+ Kursiv geschrieben)
  - S .. Static
  - F .. Final
  - C .. Constructor

# ObjectAid – Generics

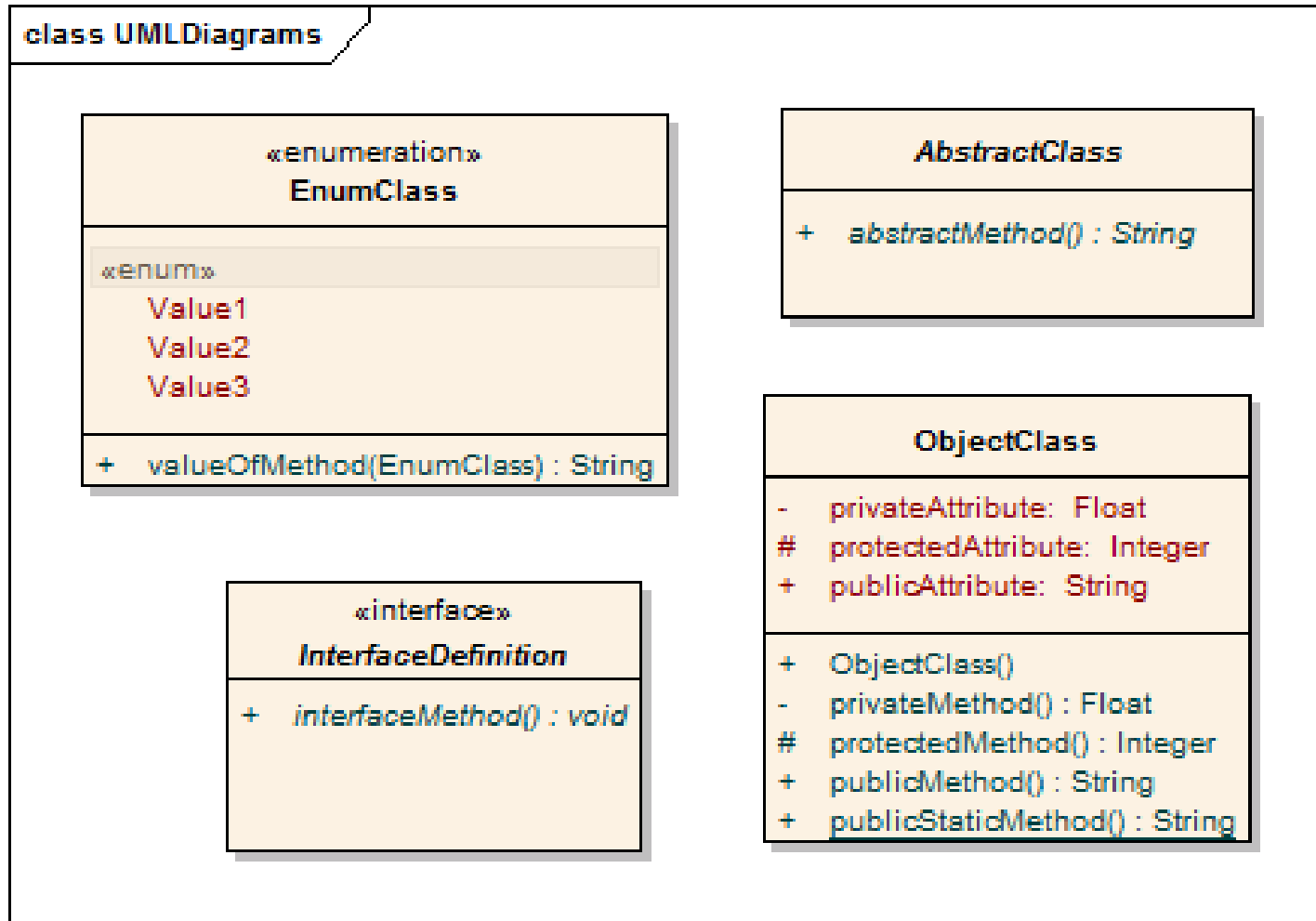


# ObjectAid - Generics

```
public interface GenericInterface<INTERFACETYPE> {  
    // Konstruktor  
    public void doAction(INTERFACETYPE param);  
    public INTERFACETYPE returnCalculatedValue();  
}
```

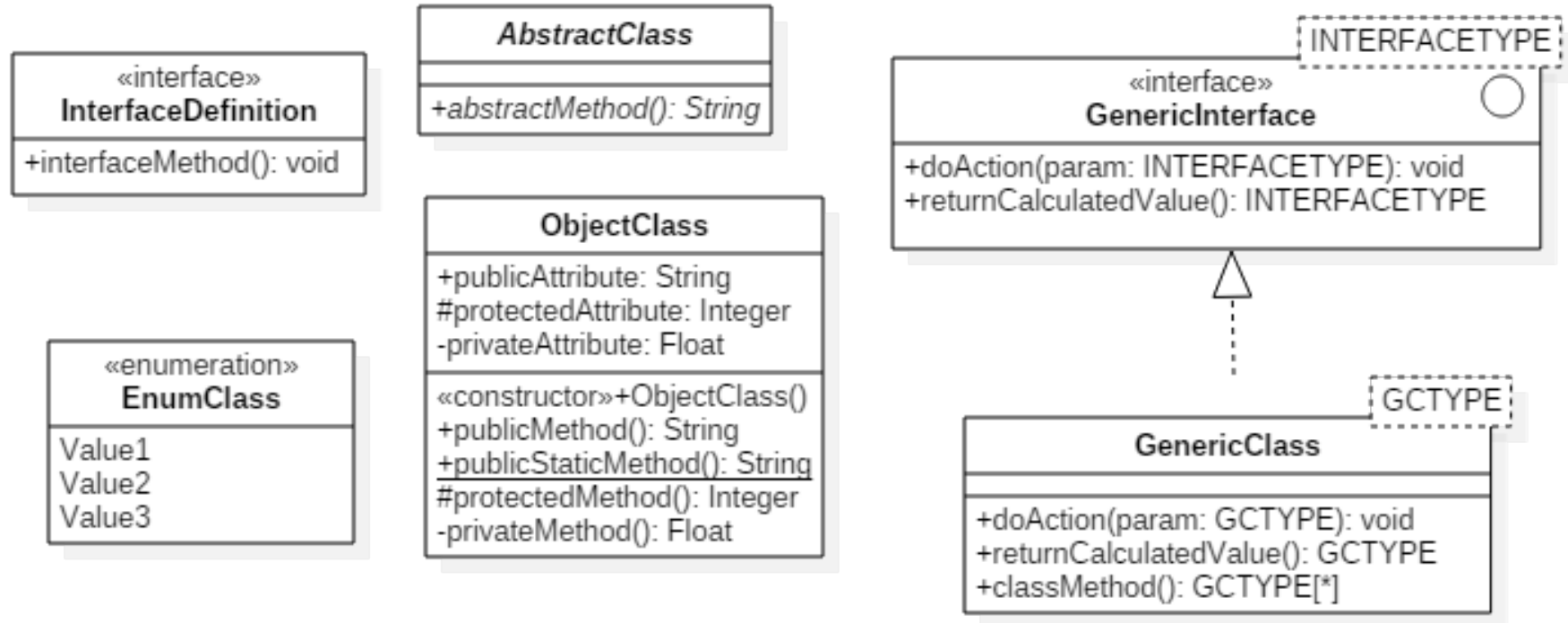
```
public class GenericClass<GCTYPE> implements  
    GenericInterface<GCTYPE> {  
  
    public void doAction(GCTYPE param) {  
  
    }  
  
    public GCTYPE returnCalculatedValue() {  
  
    }  
  
    public Collection<GCTYPE> classMethod() {  
    }  
}
```

# Enterprise Architect





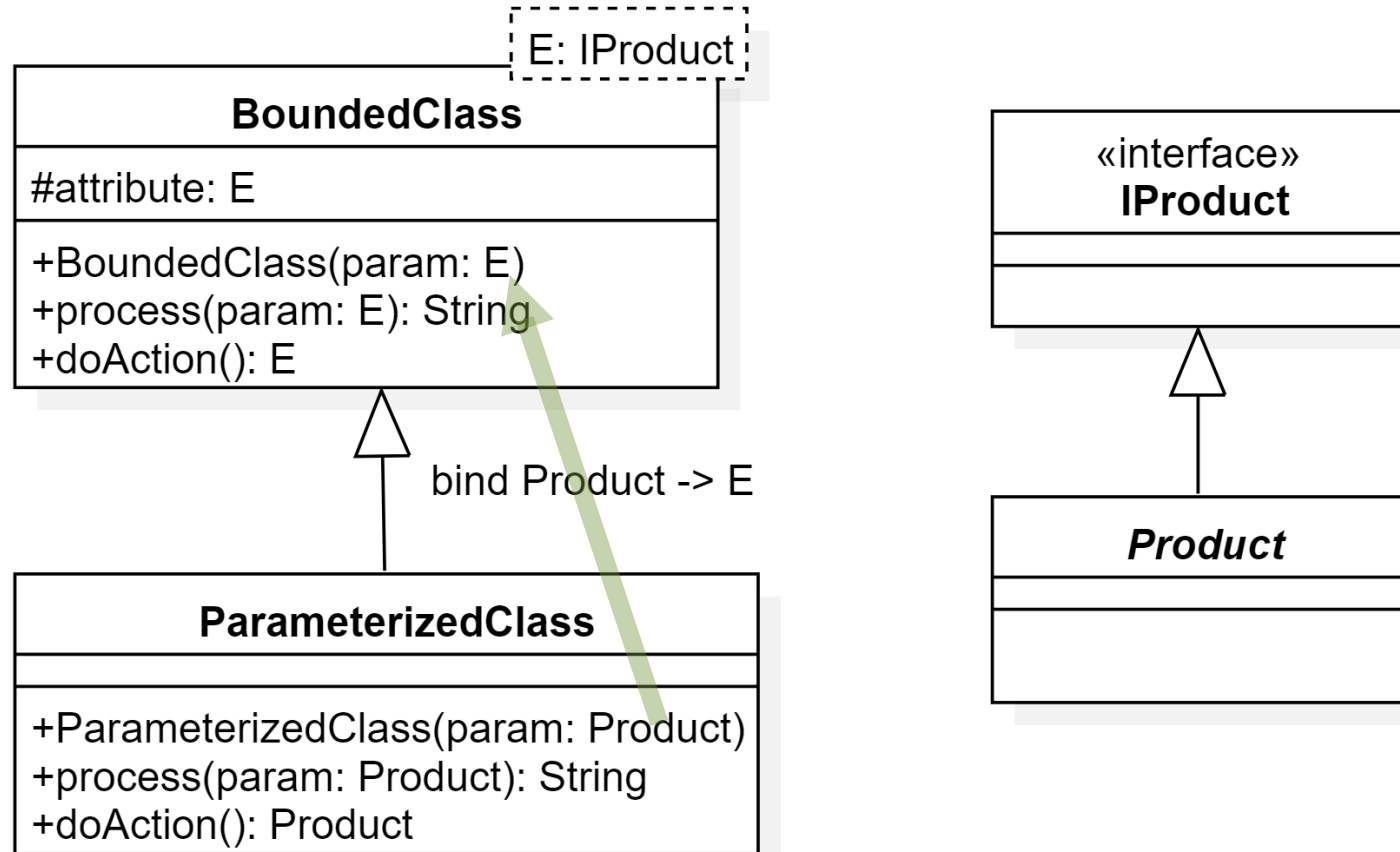
# StarUML





# Spezialisierung mit Typparametern

# UML - Spezialisierung mit Typparametern



- Typparameter E wird auf Product festgelegt
- **Optional:** Angabe des Bindings

# UML - Spezialisierung mit Typparametern

- Einschränkung eines Typparameters (obere Schranke) einer Klasse auf einen Typ.

UML Diagramm      `E : TYPE`

- Bei abgeleiteten Klassen kann das Binding durch die Ersetzung der Typen abgelesen werden.
- Optional kann das Binding auch im Klassendiagramm angegeben sein.

UML Diagramm      `bind TYPE -> E`

# UML - Spezialisierung mit Typparametern

```
public class BoundedClass<E extends IProduct> {  
  
    protected E attribute;  
  
    public BoundedClass(E param) {  
        // Konstruktor  
    }  
  
    public String process(Product param) {  
        // Methodenrumpf  
    }  
  
    public Product doAction() {  
        // Methodenrumpf  
    }  
}
```



# UML - Spezialisierung mit Typparametern

```
public class ParameterizedClass implements
    BoundedClass<Product> {

    // Typparameter E ist festgelegt mit Product

    public ParameterizedClass(Product param) {
        super(param);
        // Konstruktor
    }

    public String process(Product param) {
        // Methodenrumpf
    }

    public Product doAction() {
        // Methodenrumpf
    }
}
```



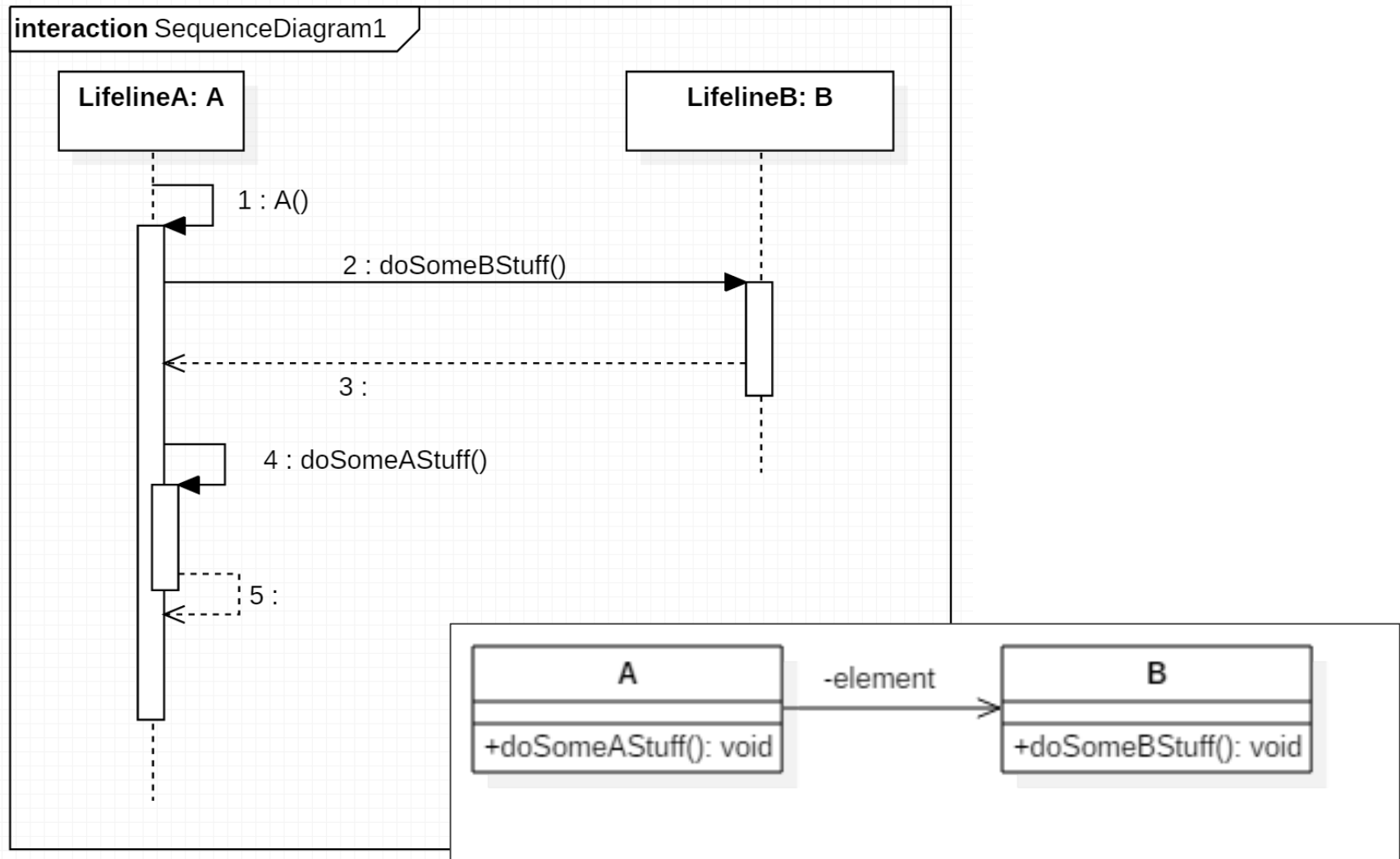
# Sequenzdiagramme

# Sequence Diagram

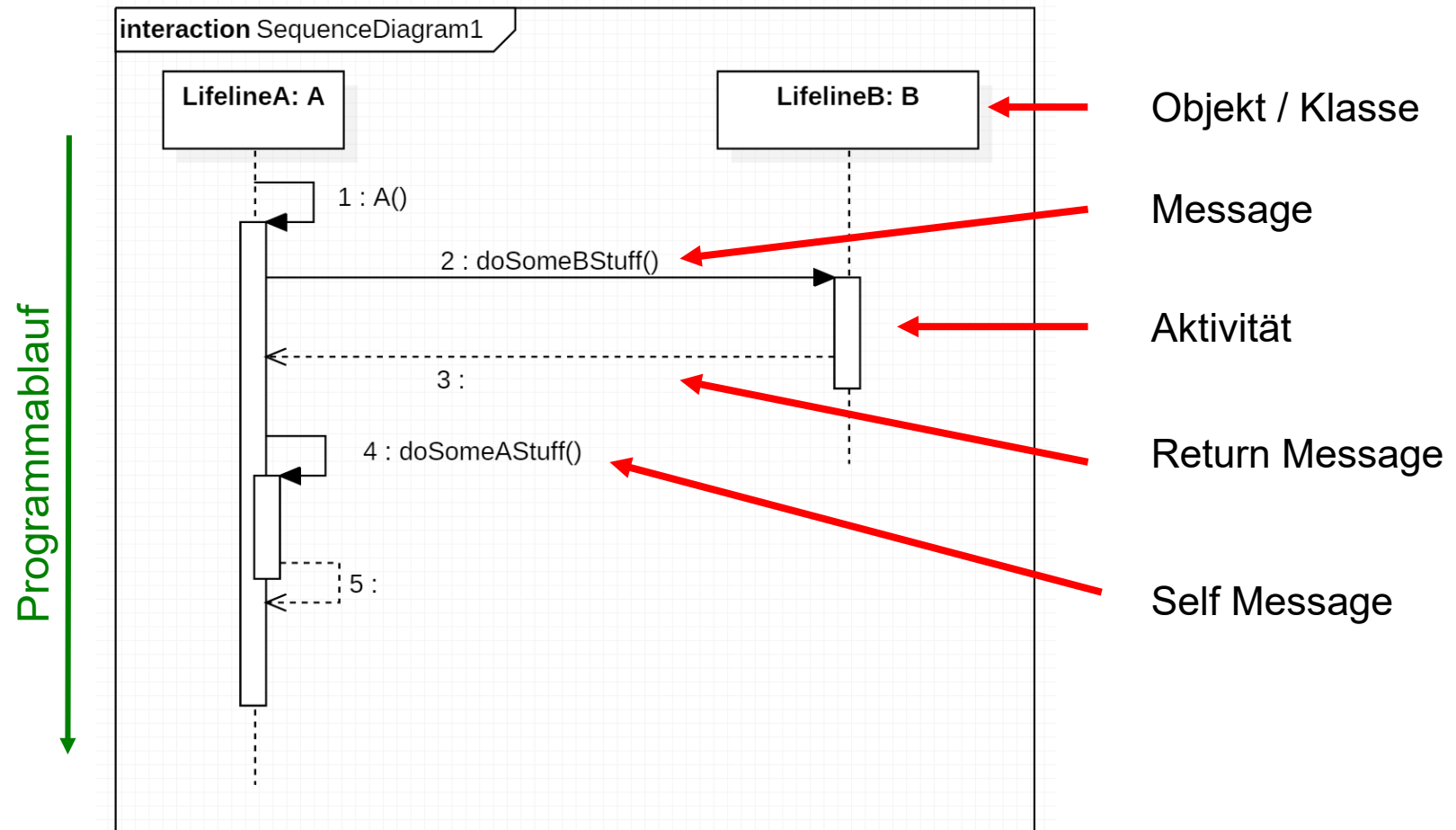
- Stellt Interaktionen dar.
  - Aber wovon ?
  - Alle möglichen Interaktionen ?
  - Nur eine gültige Abfolge ?
- Beschreibt den Austausch von Nachrichten.
  - Austausch zwischen wem?
- Zeitliche Ordnung der Ereignisse.
  - Wie erkennt man die zeitliche Ordnung ?
- In welcher Verbindung mit dem Klassendiagramm?



# Sequence Diagram



# Sequence Diagram

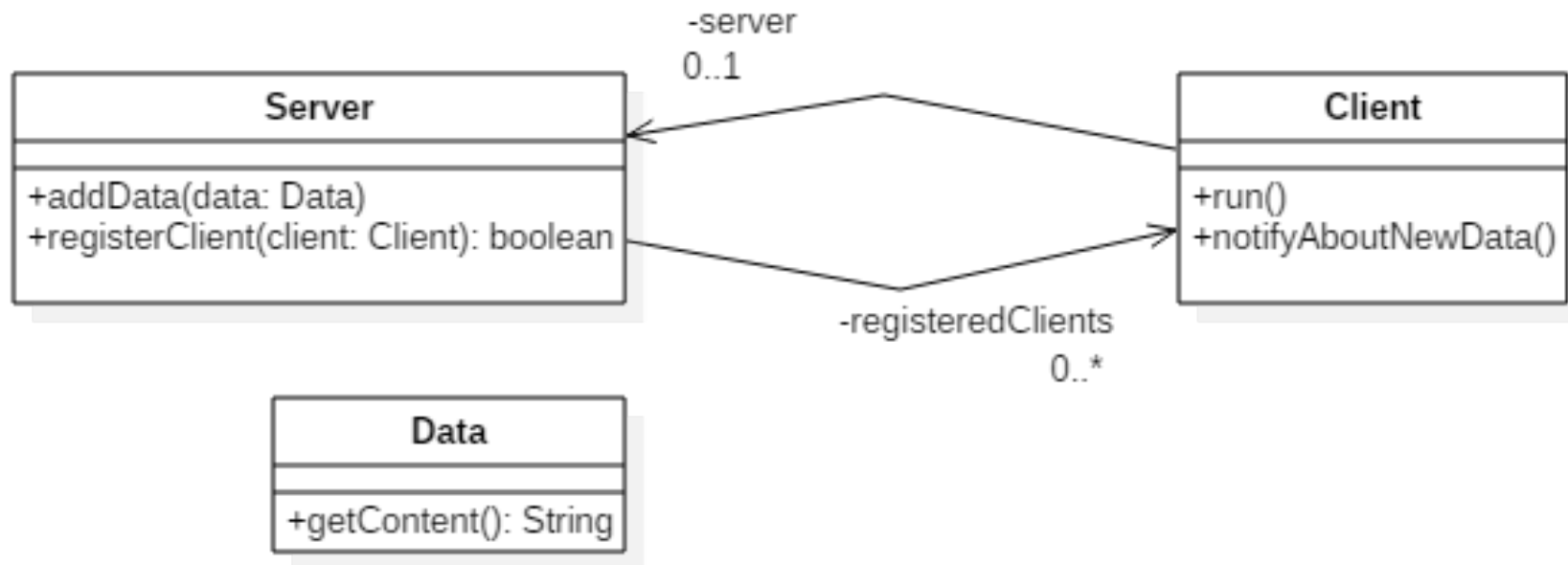


# Sprachelemente

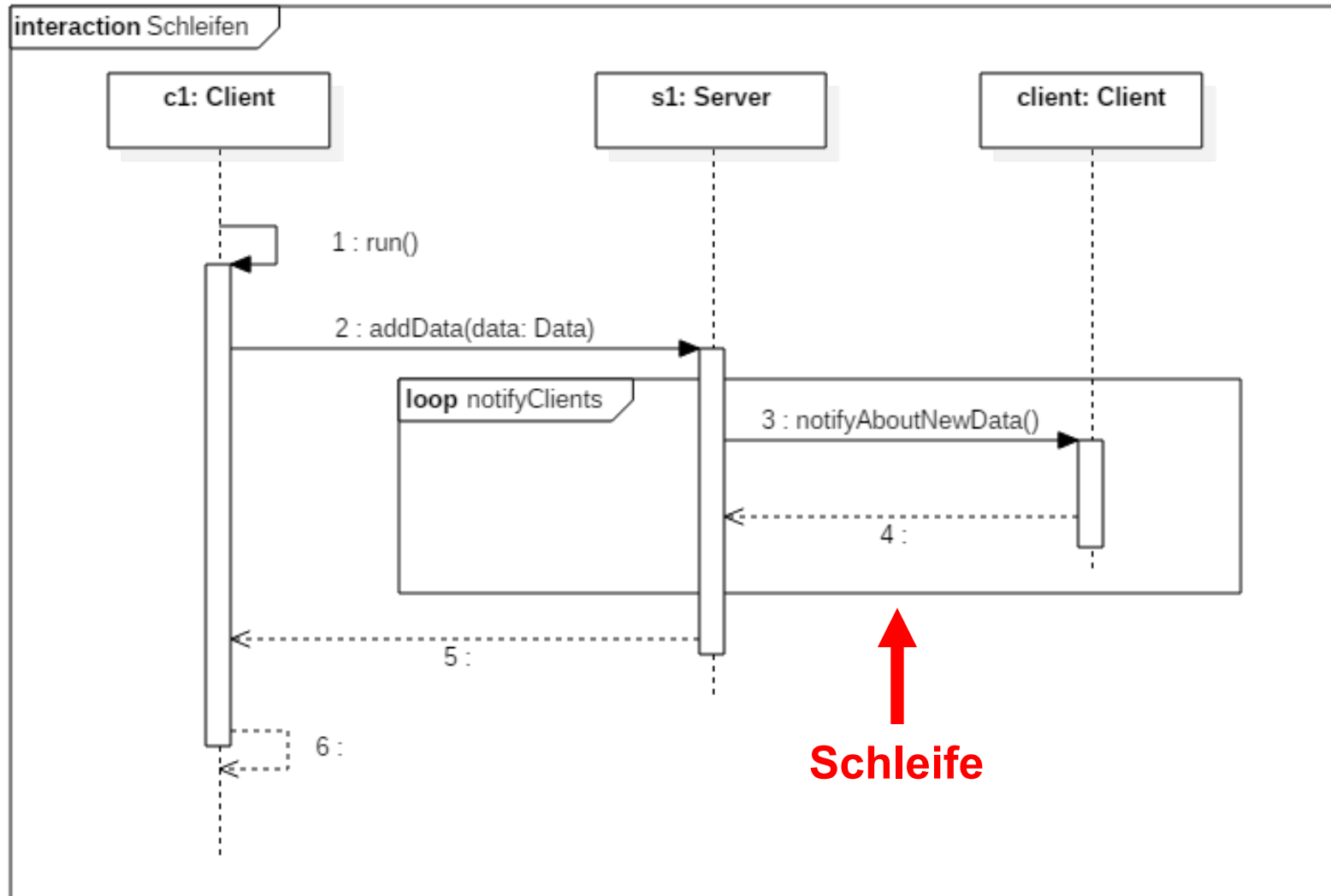
Sprachelement	Beschreibung
Loop	Soll ein Ablauf wiederholt ausgeführt werden, kann dies durch ein Loop-Fragment ausgedrückt werden.
Alt	Kennzeichnen alternative Interaktionsabläufe (Realisierung mittels switch, if , ternärer Operator, Fehlerbehandlung).
Ref	Einbinden eines anderen Sequenzdiagrammes.
Opt	Wird verwendet um optionale Schritte im Ablauf zu definieren.



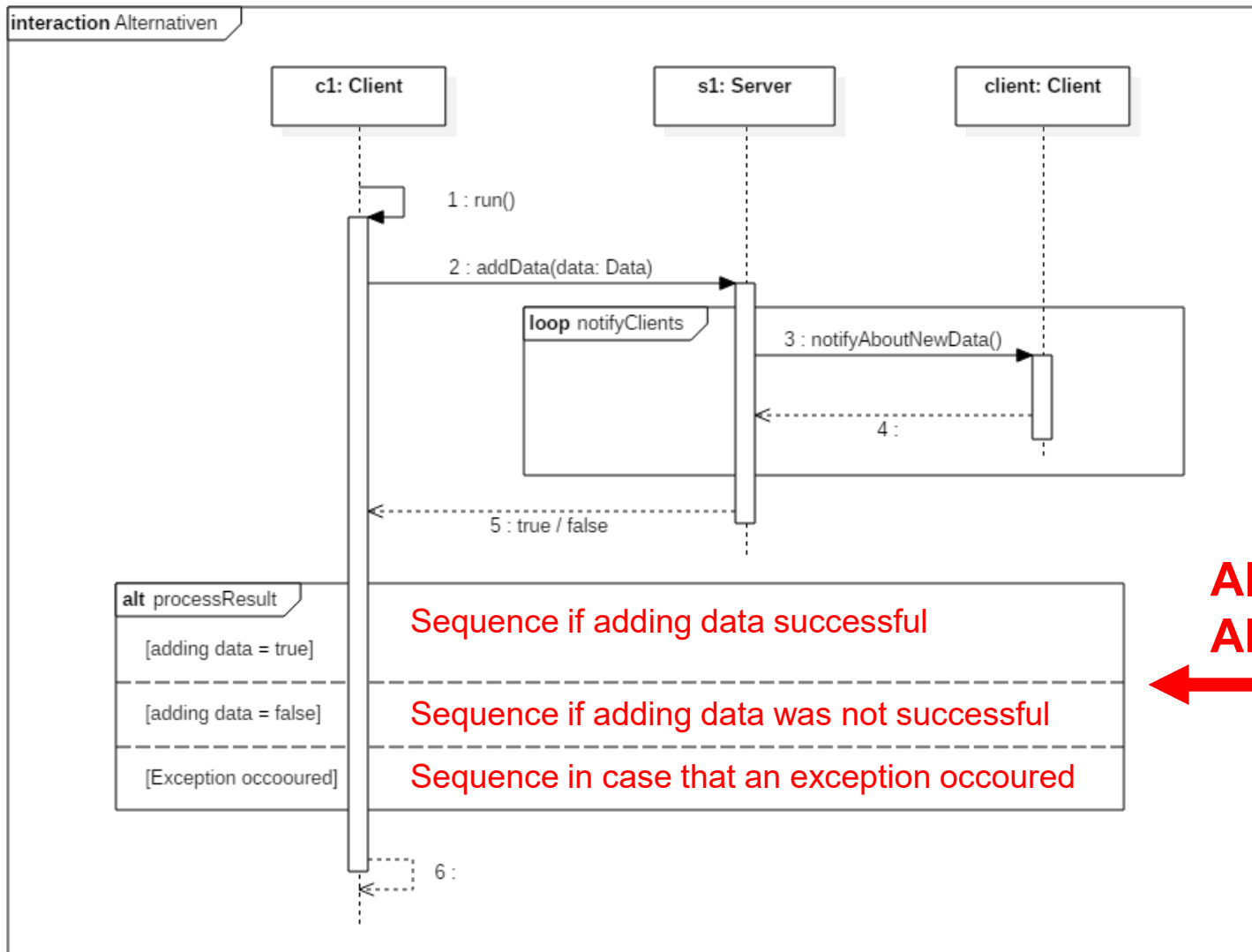
# Sprachelemente - Beispiele



# Schleifen

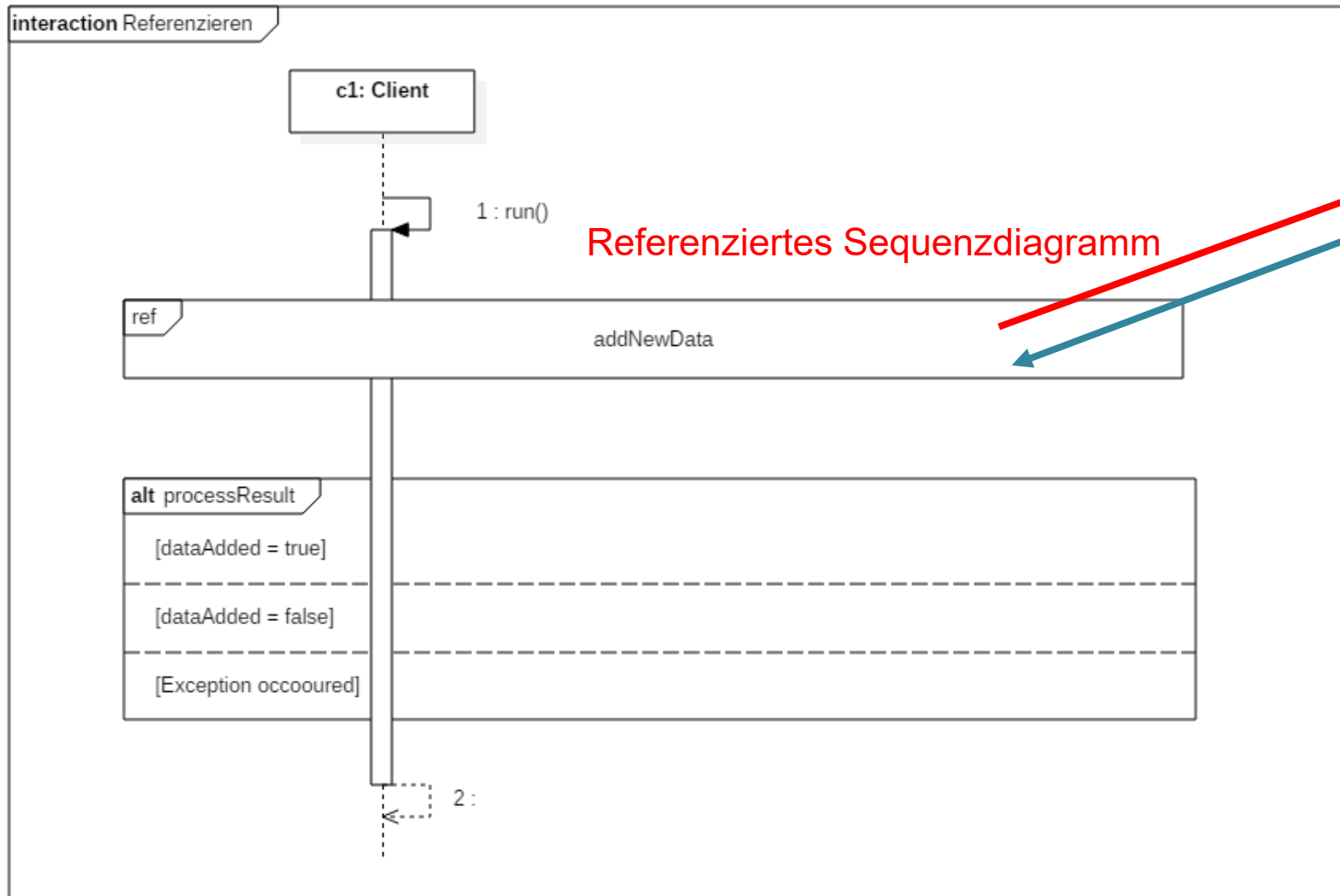


# Alternativen

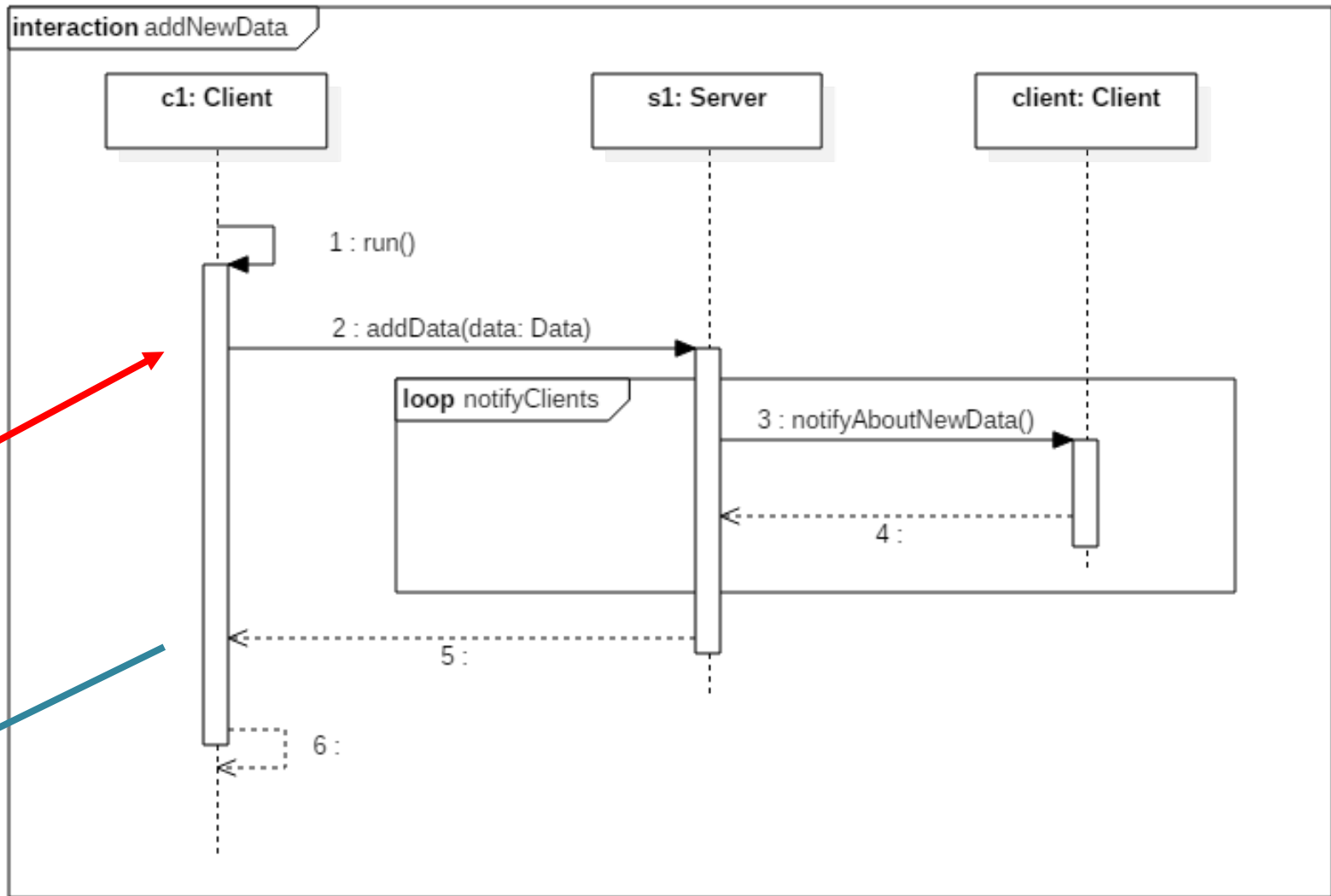


**Alternative  
Abläufe**

# Referenzierte Sequenzdiagramme



# Referenzierte Sequenzdiagramme





# Weiterführende Information

## Bücher:

UML @ Classroom, Martina Seidl, Marion Scholz, Christian Huemer, Gerti Kappel, 1. Auflage, dpunkt.verlag ISBN 978-3898647762

UML @ Work, Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, Werner Retschitzegger, 3. Auflage, dpunkt.verlag ISBN 978-3898642613

## Weblinks:

<http://www.uml.ac.at/de/>

