

# Objektorientierte Programmierung

Hochschule Bochum

WS 19/20

Dr.-Ing. Darius Malysiak

# IV. Design Patterns

## Iterator Pattern: Verhaltensmuster

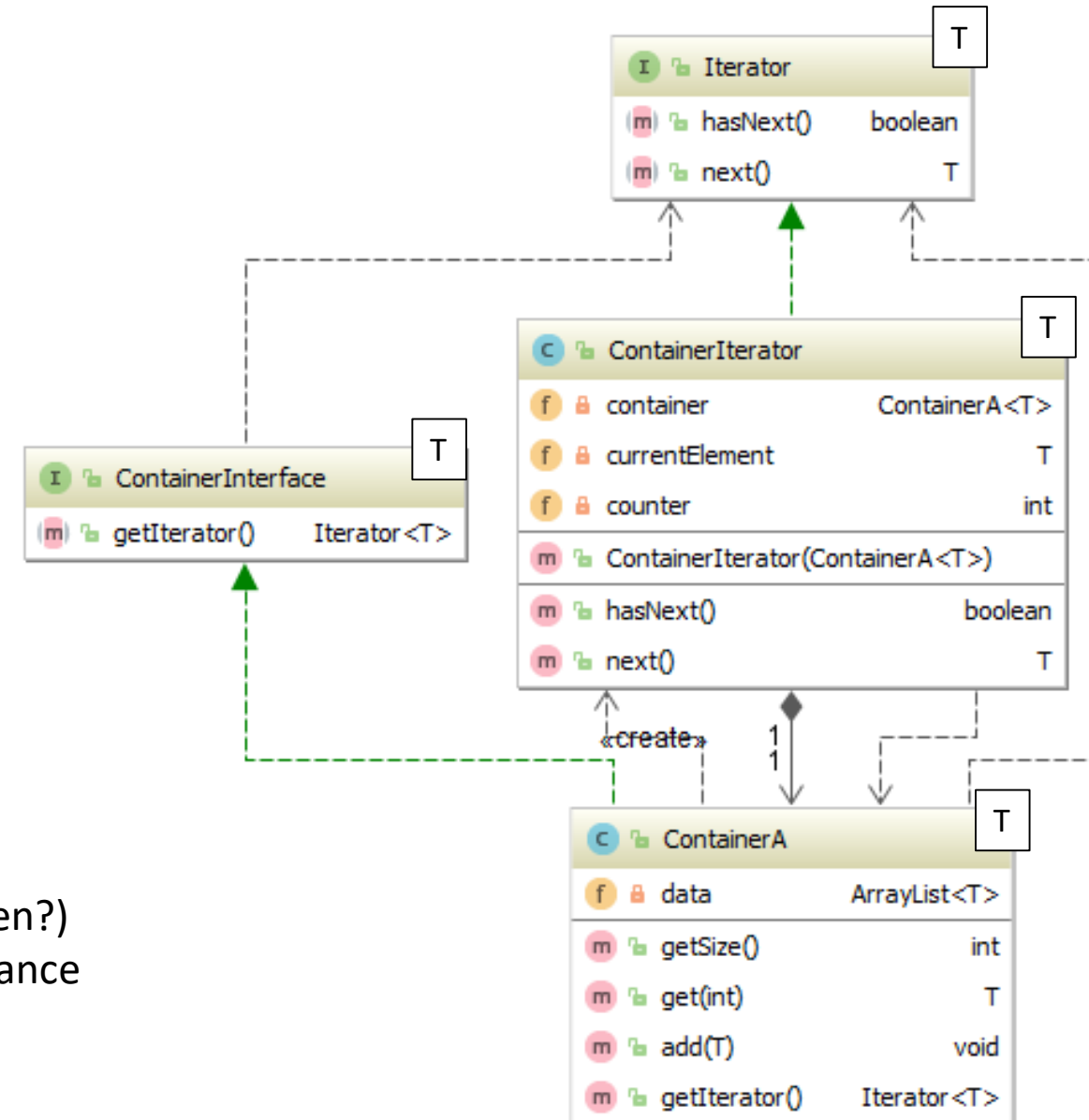
**Ziel:** Generisches Traversieren eines Containers

### 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 Performance nötig.

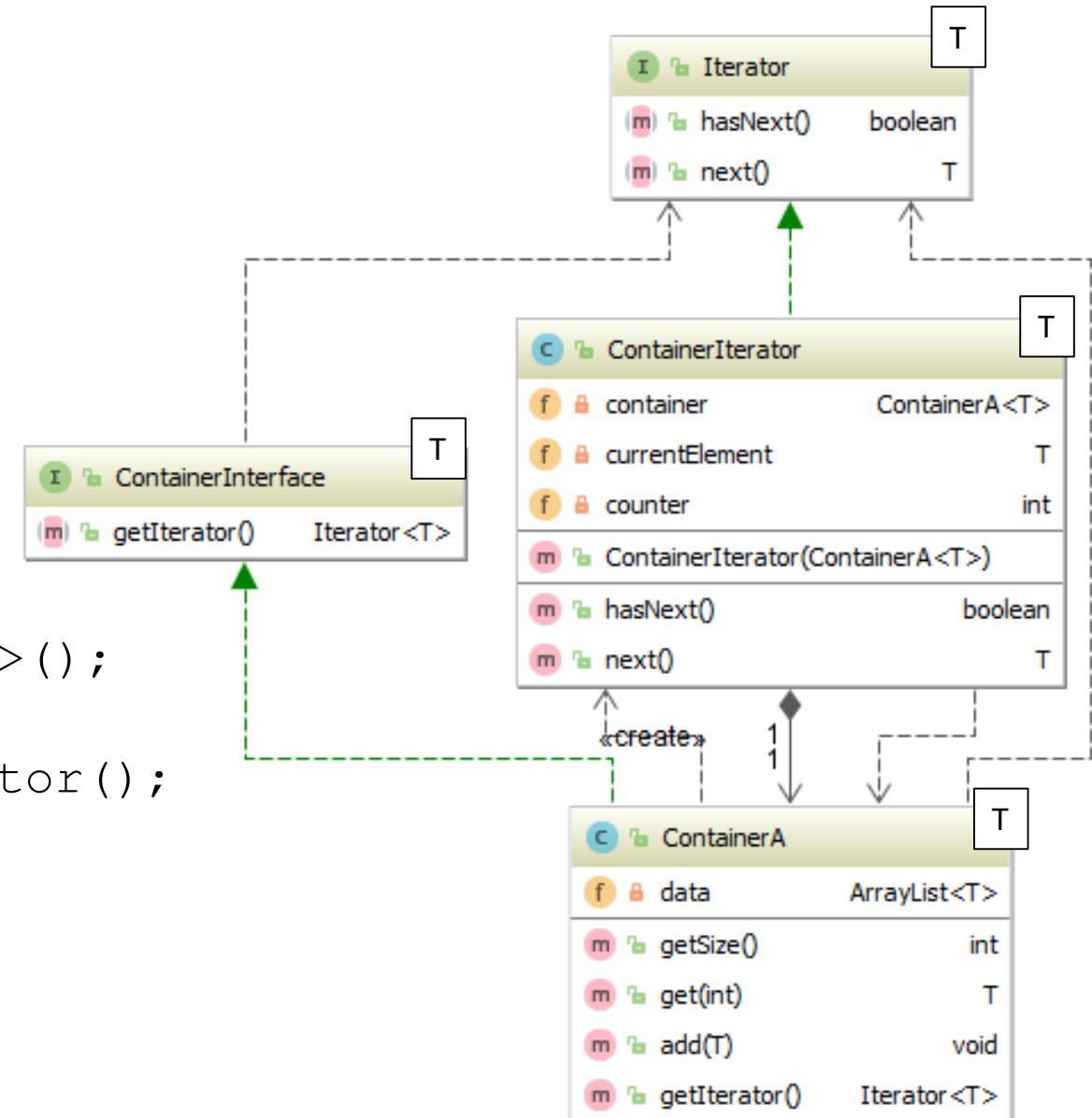


# IV. Design Patterns

**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);  
}
```



# IV. Design Patterns

**Iterator Pattern:** Verhaltensmuster

**Ziel:** Generisches Traversieren eines Containers

Prüfe ob bereits alle Elemente  
traversiert wurden.



Gebe das aktuelle Element zurück  
und inkrementiere den Zählindex.



```
public class ContainerIterator<T> implements
Iterator<T> {
    private ContainerA<T> container;
    private T currentElement;
    private int counter=0;

    public ContainerIterator(ContainerA<T> container)
    {
        this.container = container;
    }

    @Override
    public boolean hasNext() {
        if(counter<container.getSize())
        {
            return true;
        }
        return false;
    }

    @Override
    public T next() {
        currentElement = container.get(counter);
        counter++;
        return currentElement;
    }
}
```

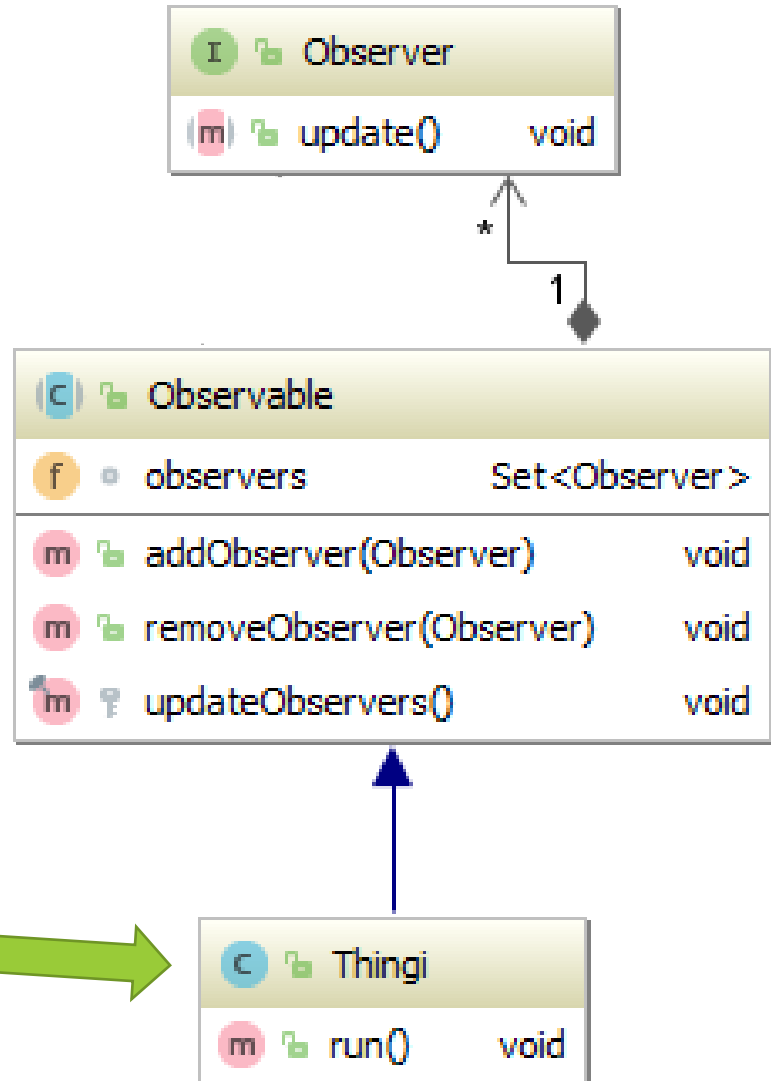
# IV. Design Patterns

## Observable Pattern: Verhaltensmuster

**Ziel:** Monitoring von Objekten auf Zustandsveränderungen

Jede Observable Instanz  
registriert bel. viele Observer

Zu überwachende Klasse  
ruft „updateObservers“ auf.

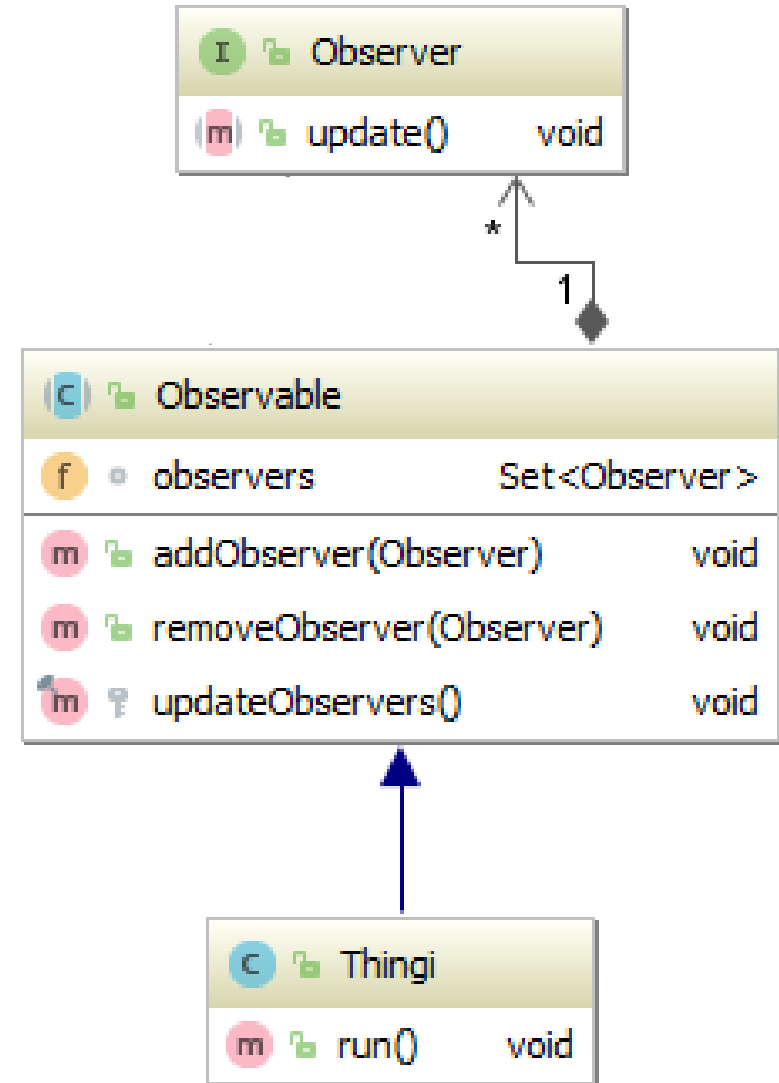


# IV. Design Patterns

## Observable Pattern: Verhaltensmuster

**Ziel:** Monitoring von Objekten auf Zustandsveränderungen

```
public abstract class Observable {  
    Set<Observer> observers = new HashSet<>();  
    public void addObserver(Observer o)  
    {  
        observers.add(o);  
    }  
    public void removeObserver(Observer o)  
    {  
        observers.remove(o);  
    }  
    final protected void updateObservers()  
    {  
        observers.stream().forEach(observer ->  
                                     {observer.update();});  
    }  
}
```



# IV. Design Patterns

## Observable Pattern: Verhaltensmuster

**Ziel:** Monitoring von Objekten auf Zustandsveränderungen

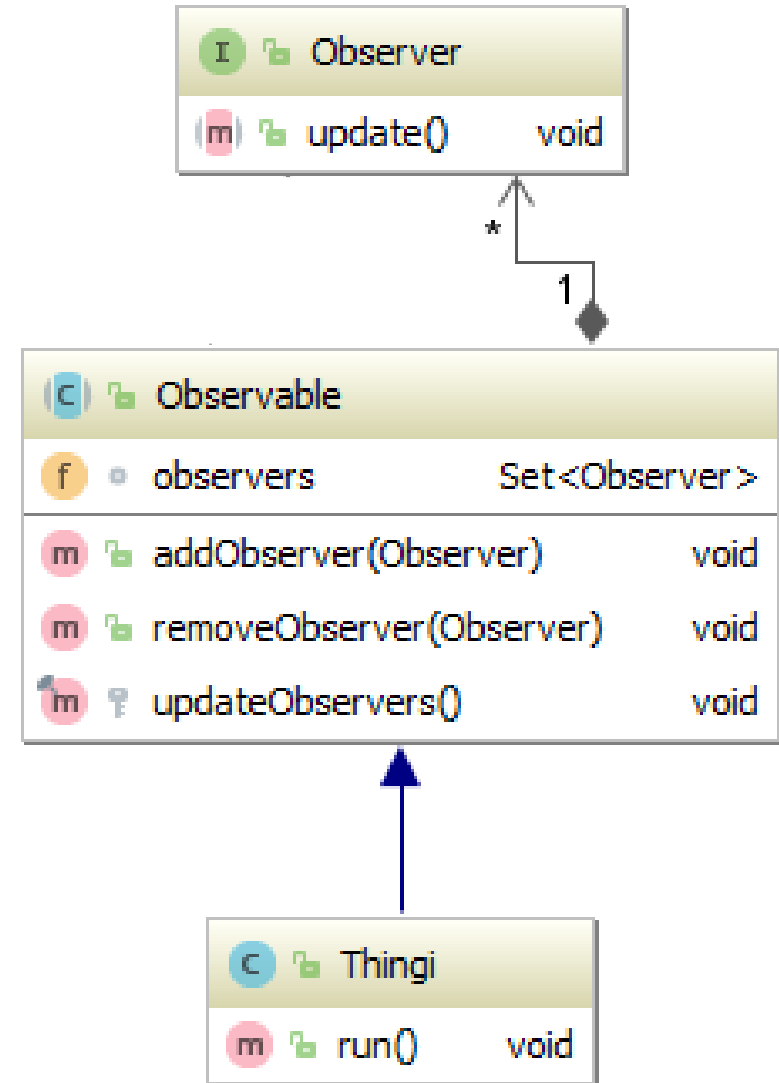
### Vorteile:

- Lose Kopplung zwischen Objekten (keine harten Verweise auf Observable Instanzen im Modell nötig, Observer stellen Koppel-elemente dar.)
- 1:N Kopplungen trivial abbildbar.

### Nachteile:

- Jede Observable Instanz muss initiativ ihre Observer informieren.
- Speicherlecks möglich (Lapsed Listener Problem)

Wann kann ein Observer garbage-collected werden?  
Observer sind in Observables registriert!




# IV. Design Patterns

## Observable Pattern: Verhaltensmuster

**Ziel:** Monitoring von Objekten auf Zustandsveränderungen

```
public class Thingi2 extends Observable
implements Runnable {
    @Override
    public void run() {
        setChanged();
        this.notifyObservers();
    }
}
```

Boolean flag true setzen;  
wird über ‚hasChanged()‘  
vom Observer abgefragt



Bereits in Java enthalten!

```
Thingi2 t2 = new Thingi2();
t2.addObserver((obj, arg) -> {System.out.println("observer3");});
t2.addObserver((obj, arg) -> {System.out.println("observer4");});
new Thread(t2).start();
```



# IV. Design Patterns

## Dependency Injection: Erstellungsmuster

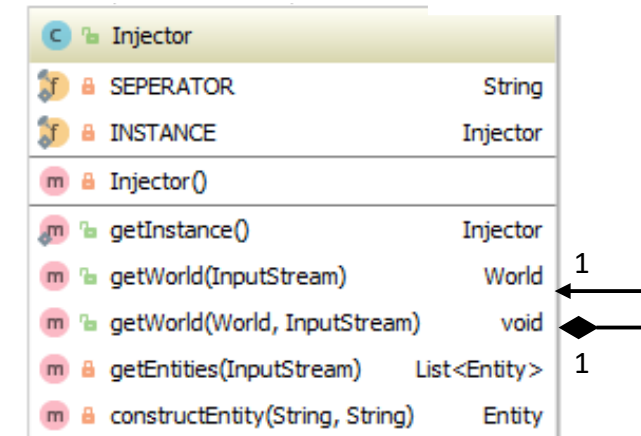
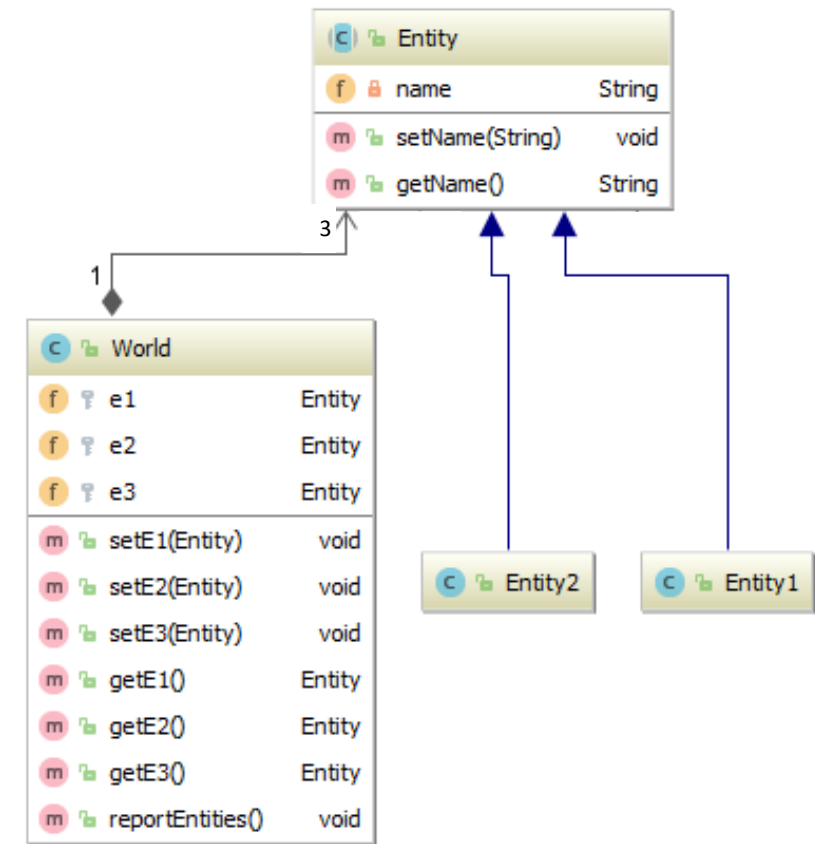
### Ziele:

- Anwendung unabhängig von der Erstellung der Objekte halten (=> Anwendung erstellt Objekte nicht selbst)
- Objekte sollen dennoch konfigurierbar sein.
- Management von Konfigurationen soll möglich sein.

### Konfiguration: Config.txt

```
solutions.exercise7.Entity2;somename  
solutions.exercise7.Entity1;anotherone  
solutions.exercise7.Entity1;testing
```

World Instanz mit 3 Entity Instanzen nach Konfiguration



# IV. Design Patterns

## **Dependency Injection:** Erstellungsmuster

### **Vorteile:**

- Trennung der ‚harten‘ Business Logic und dem Anwendungsverhalten.
- Konfiguration / Umstrukturierung der >gesamten< Anwendung möglich ohne neues Kompilieren.
- Implizite Vermeidung von Boilerplate Code durch Initialisierungs-Facilities.
- Entwicklung teilweise vereinfacht da Objekte hier als externe ‚Systeme‘ betrachtet werden können, mehrere Entwickler können parallel an diversen solcher Systeme arbeiten.
- Lockert die Kopplung innerhalb der Anwendung.

### **Nachteile:**

- Debugging wird ungleich erschwert da nicht eindeutig klar ist, welche Klassen injiziert werden.
- IDE Support unumgänglich zur Navigation / Debugging da hier ein extremer Gebrauch von Reflections auftritt.
- Externe Konfiguration wird essentieller Bestandteil der Gesamtanwendung.
- Abhängigkeit von Management-Frameworks wie Spring durch Verwendung von speziellen Annotationen.

# IV. Design Patterns

## Beans / POJOs:

### Def.:

Eine serialisierbare Klasse mit Default Konstruktor, welche mehrere Attribute kapselt und diese per Getter/Setter zugänglich macht wird als JavaBean oder Bean bezeichnet.

### Def.:

Ein Objekt eine Klasse wird als Plain Old Java Object (POJO) bezeichnet falls die Klasse keinerlei externe Abhängigkeiten hat um Objekte instanziierten zu können. Externe Abhängigkeiten sind hierbei

- Annotationen aus externen Frameworks.
- Klassen / Interfaces aus externen Frameworks.

# IV. Design Patterns

## Inversion of Control

### Definition:

Wenn der gesamte oder partielle Programmfluss eines Programms von externen Frameworks / Systemen gesteuert wird, so sprechen wir von Inversion of Control (IC). Hierbei ist der zeitliche Punkt des Auftretens einer solchen externen Kontrolle irrelevant.

Mit anderen Worten, das Programm wird ganz oder teilweise von externen Elementen gesteuert.

Inversion of Control ist die direkte Verallgemeinerung von Dependency Injection


# IV. Design Patterns

## Inversion of Control

Früher wurden Beans in Form von externen XML Dateien für Dependency Injection bereitgestellt:

- XML ist sehr ausdrucksstark, jedoch für Menschen schwer lesbar.
- Dynamische Anpassung von XML Dateien benötigt XML Parser / Modifier.
- Selbst kleine Änderungen benötigen Änderung der XML Datei.
- Beans müssen manuell durch XML Dateien erstellt werden.

Java Spring:

- Open Source Framework für Annotation Driven Development von JEE.
- Industrie Standard.
- Mit Hilfe von Erweiterungen wie z.B. Spring Boot (keine xml Dateien zur Erstellung von Beans) wird die Applikationsentwicklung rapide Beschleunigt.
- Konfiguration von Beans (d.h. deren Attributen oder überhaupt der gewünschten Klasse) über Annotationsparameter.
- Aspektorientiert Programmierung (AOP)  Später mehr!



[https://de.wikipedia.org/wiki/Spring\\_\(Framework\)#/media/File:Spring\\_Logo.png](https://de.wikipedia.org/wiki/Spring_(Framework)#/media/File:Spring_Logo.png)

# IV. Design Patterns

## Inversion of Control

Bsp.:

```
@Configuration
public class InfluxDBConfig {
    @Value("${DATABASE_NAME}")
    public String dbName = "history";

    @Bean
    public InfluxDB influxDB() {
        InfluxDB influxDB =
            InfluxDBFactory.connect(influxDBHost,
                                   influxUser, influxPassword);
        return influxDB;
    }
}
```

Automatische Zuweisung  
zur Laufzeit

```
@Slf4j
public class DatabaseBeans {

    @Autowired
    InfluxDB influxDB;

    @Autowired
    public SimpleDateFormat dateTimeFormatter;
}
```

## IV. Design Patterns

Wir bauen eine Dependency Injection Facility

