

# Verteilte Systeme

Prof. Dr. Martin Becke

CaDS - HAW Hamburg

Version 0.9

# Inhalt

- 1 Supporting Patterns II
  - Dependency Injection
  - Wrapper und Adapter
  - Interceptor
  - Watchdog
  - Fassade
  - Pipeline
  - Master-Worker

# Dependency Injection

## Idee

- ▶ Abhängigkeiten eines Objekts wird von außen bereitgestellt
- ▶ Code wird modularer, testbarer und wartbarer
- ▶ Gleichen Vorteile wie das Factory Pattern
- ▶ Führt zusätzliche Komplexität ein
- ▶ Leistung kann reduziert werden, wenn Abhängigkeiten über das Netzwerk transportiert werden muss
- ▶ Kann Sicherheitsprobleme einführen
- ▶ Problematisch im Debugging

# Dependency Injection

## Umsetzung in Code I

---

```
public interface BulbService {  
    void doLight();  
}
```

---

Listing 1 – Schnittstelle für die Abhängigkeit

# Dependency Injection

## Umsetzung in Code II

---

```
public class Bulb implements BulbService {  
    public void doLight() {  
        System.out.println("Do cool Stuff");  
    }  
}
```

---

Listing 2 – Schnittstellenimplementierung

# Dependency Injection

## Umsetzung in Code III

---

```
public class ControllerOnPI {  
    private BulbService service;  
  
    public ControllerOnPI(BulbService service) {  
        this.service = service;  
    }  
  
    public void doControl() {  
        service.doLight();  
    }  
}
```

---

Listing 3 – Dependency

# Dependency Injection

## Umsetzung in Code IV

---

```
public class Main {  
    public static void main(String[] args) {  
        BulbService bulb = new Bulb();  
        ControllerOnPI c = new ControllerOnPI(bulb);  
  
        c.doControl();  
    }  
}
```

---

Listing 4 – Die DI Main

# Dependency Injection

## Reflection

- ▶ Dependency Injection (DI) wird häufig mit Reflection umgesetzt
- ▶ Struktur und Verhalten zur Laufzeit analysieren
- ▶ Eigenschaften und Funktionalitäten können zur Laufzeit verändert werden
- ▶ In vielen Programmiersprachen unterstützt, z. B. in Java, C#, Python und JavaScript



# Wrapper und Adapter

## Idee

- ▶ Strukturpattern
- ▶ Etabliert Interoperabilität
- ▶ Beide Muster sind ähnlich, es gibt aber Unterschiede

# Adapter

## Grundlagen

- ▶ Für Komponenten mit unterschiedlichen Schnittstellenanforderungen
- ▶ Etabliert Interoperabilität
- ▶ Kann unterschiedliche Protokollen oder Datenformaten miteinander verbinden
- ▶ Bestehende Komponente an eine neue Schnittstelle anpassen (legacy oder Zulieferung)

# Wrapper

## Grundlagen

- ▶ Definiert eine neue Klasse, die eine vorhandene Komponente umschließt
- ▶ Verändert oder erweitert Funktion

# Interceptor

## Idee

- ▶ Steuert Kommunikation zwischen Komponenten
- ▶ Eingehende und ausgehende Nachrichten werden abgefangen
- ▶ Möglicher Einsatz : Sicherheitsüberprüfungen, Protokollierung, Leistungsüberwachung und/oder Fehlerbehebung

# Interceptor Pattern

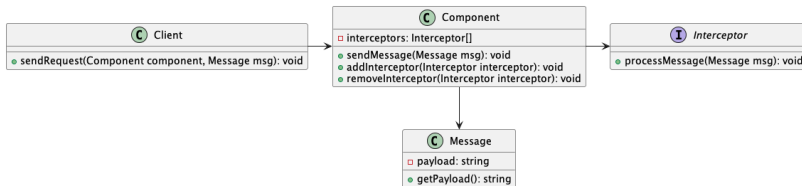


Figure – Interceptor Pattern

# Interceptor

## Sequenz

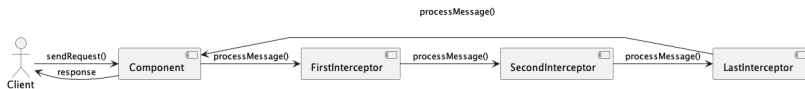


Figure – Interceptor Pattern Sequenz

# Watchtog

## Watchdog

- ▶ Überwachung von Ressourcen oder Prozessen in einem System
- ▶ Erhöht Stabilität und Zuverlässigkeit
- ▶ Zwei Hauptkomponenten : Watchdog und überwachtes System
- ▶ Frühzeitige Erkennung und Behebung von Problemen
- ▶ Anwendbar in verschiedenen Bereichen (z.B. eingebettete Systeme, verteilte Systeme)
- ▶ Trennung der Verantwortlichkeiten für optimale Effektivität

# Fassade

## Idee

- ▶ Vereinfachte Schnittstelle für den Zugriff
- ▶ Reduktion der Komplexität



# Pipeline

## Idee

- ▶ Für komplexe Verarbeitungsprozesse
- ▶ Sequentielle Ausführung
- ▶ Spezialisierten Modul (Fetch, Decode, Execute Pipeline)

# Master-Worker

## Idee

- ▶ Speziell für verteilte Systeme entwickelt
- ▶ Zentrale Einheit (Master) welcher die Kontrolle über mehrere untergeordnete Einheiten (Worker) hat
- ▶ Arbeitslast effizient auf mehrere Prozessoren oder Knoten zu verteilen