

# Verteilte Systeme

Prof. Dr. Martin Becke

CaDS - HAW Hamburg

Version 0.9

# Inhalt

- 1 Supporting Patterns I
  - Model Viewer Controller
  - Vertreter
  - Observer
  - Callback
  - Singleton
  - Factory

# Model Viewer Controller (MVC)

## Idee

- ▶ Sowohl Entwurfsmuster, also auch Designentwurf
- ▶ Hier Konzentration auf MVC-Architekturmuster
- ▶ Besteht aus drei Hauptkomponenten
- ▶ Klare Trennung im Modell kann Aufwände verursachen (JavaFX) -> Siehe Beispiel

# Model View Controller (MVC)

## Struktur

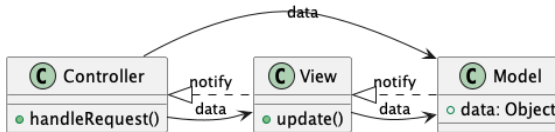


Figure – Beispiel eines Basis MVC Architekturmodells

# Model View Controller (MVC)

## Fallbeispiel Led-Lampe

- ▶ Mobile-App (Android Node)
- ▶ Node.js Instanz (PI)
- ▶ Smarte Lampe (Leuchtmittel mit ESP32)

# Model View Controller (MVC)

## Struktur

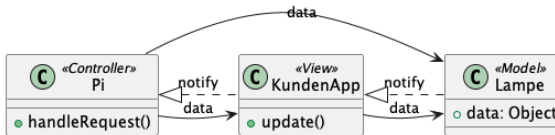


Figure – MVC Architekturmusters im Fallbeispiel mit Sterotypen

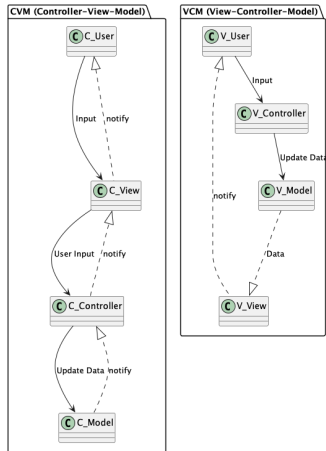


Figure – Mögliche Varianten vom MVC in Schichten gedacht

```
public class JavaFXUserInputExample extends Application {  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) {  
        Button button = new Button("Click me!");  
  
        // Registering an EventHandler for the button click  
        button.setOnAction(event -> {  
            System.out.println("Button clicked!");  
        });  
  
        StackPane root = new StackPane();  
        root.getChildren().add(button);
```



# Proxy

## Idee

- ▶ Zwischen Consumer und Provider platziert
- ▶ Ein Platzhalter, ein Vermittler
- ▶ Kann mit Caching Netzwerklatenz reduzieren
- ▶ Kann die Sicherheit erhöhen

# Proxy

## Design als Pattern

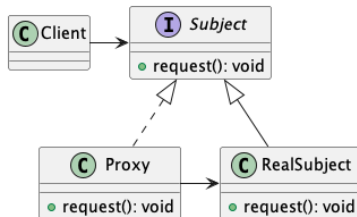


Figure – Proxy Pattern

# Proxy

## Typen

- ▶ Forward Proxy
- ▶ Reverse Proxy
- ▶ Caching Proxy
- ▶ Load Balancing Proxy

# Broker

## Idee

- ▶ Zwischen Consumer und Provider platziert
- ▶ Ein Vermittler mit erweiterten Funktionen
- ▶ Kann Nachrichten filtern, transformieren und aggregieren
- ▶ Unterstützt mehrere Protokolle
- ▶ Kann über eine Warteschlange für eine asynchrone Auslieferung verfügen

# Broker

## Design als Pattern

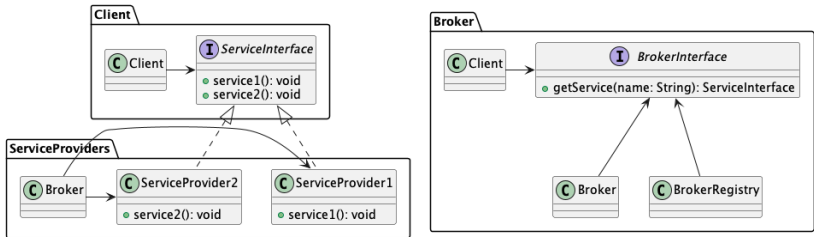


Figure – Broker Pattern

# Broker

## Typen

- ▶ Forwarding-Broker
- ▶ Handle-Driven-Broker

# Trader

## Idee

- ▶ Zwischen Consumer und Provider platziert
- ▶ Ein Vermittler mit eigener Entscheidungskompetenz
- ▶ Kann Katalog von Diensten nach Kriterien anbieten

# Observer

## Idee

- ▶ Lösung für unmittelbare Kommunikation
- ▶ 1 :n-Beziehung zwischen Objekten
- ▶ Benachrichtigt, wenn sich sein Zustand ändert (Event)
- ▶ Ermöglicht eine lose Kopplung im Design
- ▶ Erhält enge Kopplung in der Kommunikation



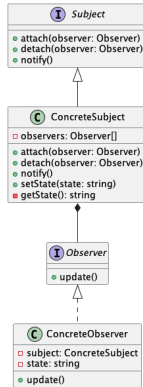


Figure – Einfaches Observer Pattern

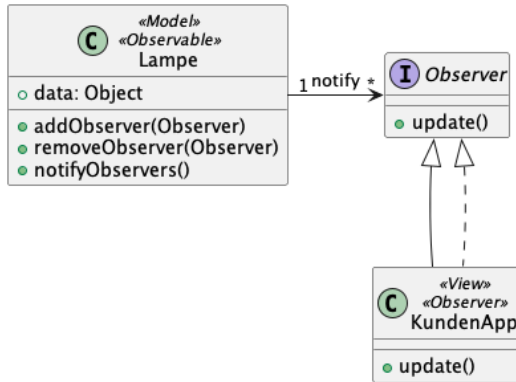


Figure – Observer Pattern als Verbindung zwischen Model und View

# Observer

## Pattern Beispiel Lampe - Verhalten

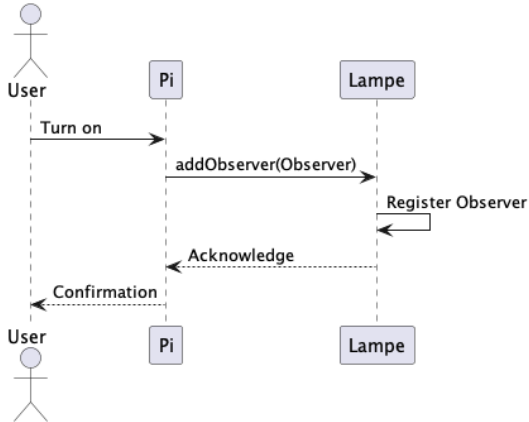


Figure – Observer Pattern als Verbindung zwischen Model und View

# Observer

Im Kontext von MVC und VS

- ▶ Nicht nur eine Design-Frage
- ▶ Technologie-abhängig
- ▶ Uni-direktionalen Kommunikation zwischen dem Subjekt und den Beobachtern
- ▶ Request-Response-Cycle unterstützt Observer nicht direkt
- ▶ WebSockets, Long Polling, HTTP2 oder HTTP3 können Optionen anbieten

# Callback

## Idee

- ▶ Funktion (Methode oder Prozedur) als Argument an eine andere Funktion
- ▶ Nach Abschluss einer bestimmten Aufgabe wird Funktion ausgeführt
- ▶ Aufrufende Funktion muss nichts vom Callback wissen
- ▶ Kommunikation unidirektional
- ▶ Callback-Pattern wird häufig in der asynchronen Programmierung verwendet
- ▶ Callback-Pattern eignet sich gut für ereignisgesteuerte Programmierung

# Callback

## Callback Pattern

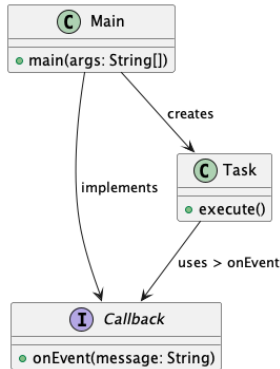


Figure – Callback Pattern

# Callback

## Callback Pattern Sequenz

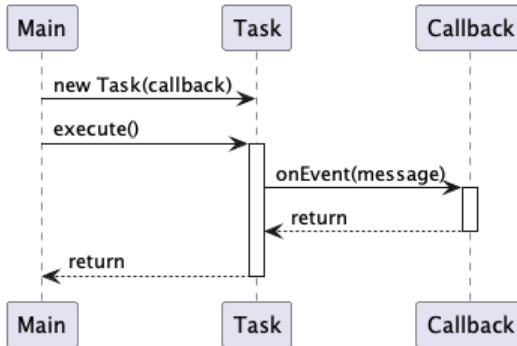


Figure – Callback Pattern Sequenz

# Singleton

## Idee

- ▶ Klasse hat nur eine einzige Instanz
- ▶ Globalen Zugriffspunkt zu dieser Instanz



# Singleton

## Pattern

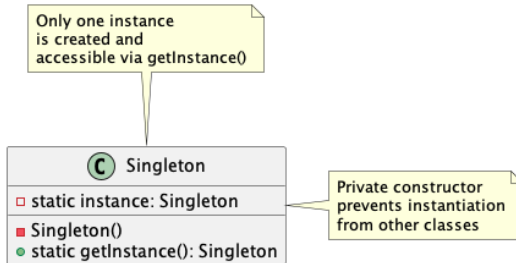


Figure – Singleton Pattern

# Singleton

## Diskussion Eignung

- ▶ Herausforderung in VS, insbesondere bei gleichzeitigen Zugriff
- ▶ Nutzen spezifischen Anforderungen
- ▶ Manchmal sinnvoll : Zentrale Verwalten von Ressourcen und die Steuerung eines globalen Zustands
- ▶ Alternativen prüfen !

# Factory

## Idee

- ▶ Objekterstellung mittels separater Klasse
- ▶ Kann Client-Code von den konkreten Implementierungsdetails entkoppeln
- ▶ Verschiedene Implementierungen können unterstützt werden
- ▶ Kann Komplexität kapseln
- ▶ Ressourcen können zentral erstellt und verwaltet werden
- ▶ Kann Load Balancing und Failover-Mechanismen implementieren

# Factory Pattern

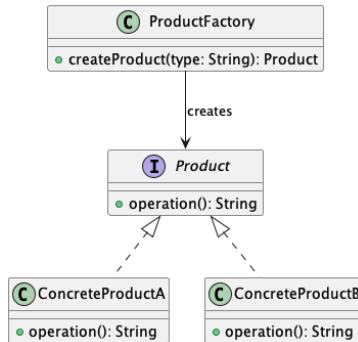


Figure – Factory Pattern

# Factory

## Sequenz

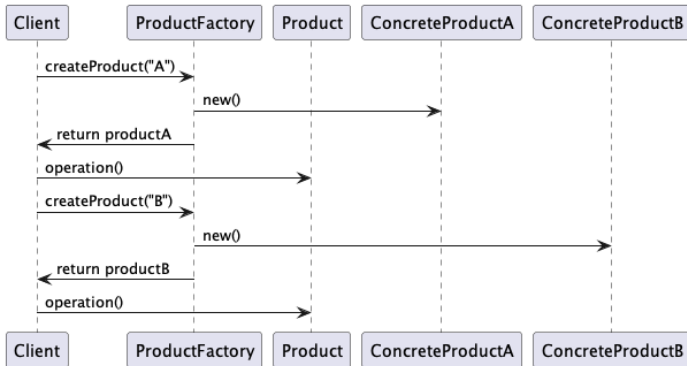


Figure – Factory Pattern Sequenz

# Factory

## Diskussion Eignung

- ▶ Code sauberer und wartbarer
- ▶ Skalierbarkeit und Fehlertoleranz verbessert
- ▶ Viele Probleme bleiben : Kommunikationslatenz, Synchronisation, SPOF, Komplexität, Testbarkeit, Overhead
- ▶ Meist besser als Singleton