# Design Patterns Workshop

Telekom Architecture Training


*Professional software architecture patterns for enterprise development*

# Workshop Agenda

# Design Patterns in der Softwareentwicklung

Design Patterns sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in der Softwareentwicklung. Sie beschreiben die Kommunikation zwischen Objekten und Klassen, die angepasst werden, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.

Jedes Pattern beschreibt ein Problem, das immer wieder in unserer Umgebung auftritt, und dann den Kern der Lösung zu diesem Problem, auf eine Weise, dass Sie diese Lösung millionenfach anwenden können, ohne sie jemals auf die gleiche Weise zu implementieren.

# Patterns Übersicht

## Erstellungsmuster

- **Singleton** - Eine einzige Instanz
- **Factory Method** - Objekterstellung delegieren
- **Abstract Factory** - Produktfamilien erstellen
- **Builder** - Komplexe Objekte schrittweise
- **Prototype** - Objekte durch Klonen erstellen

## Strukturmuster

- **Adapter** - Inkompatible Interfaces verbinden
- **Decorator** - Verhalten dynamisch erweitern
- **Facade** - Vereinfachte Schnittstelle
- **Composite** - Hierarchische Strukturen
- **Proxy** - Stellvertreter für andere Objekte

# Praxisbeispiele

Patterns in der Praxis:

- **MVC Architecture Pattern** - Model-View-Controller Trennung
- **Dependency Injection** - Lose Kopplung von Komponenten
- **Repository Pattern** - Datenzugriff abstrahieren
- **Observer** - Event Handling implementieren
- **Strategy** - Algorithmus-Auswahl zur Laufzeit

# Singleton Pattern

Das Singleton Pattern stellt sicher, dass eine Klasse nur eine Instanz hat und bietet einen globalen Zugriffspunkt darauf.

- **Zentrale Ressourcenverwaltung** - Database connections, logging
- **Thread-sichere Implementierung** - Concurrent access protection
- **Lazy Loading Unterstützung** - Instanziierung bei Bedarf
- **Einfache `getInstance()` Methode** - Konsistenter Zugriff

```
// Singleton Pattern in JavaScript
class DatabaseConnection {
  static instance = null;

  constructor() {
    if (DatabaseConnection.instance) {
      return DatabaseConnection.instance;
    }

    this.connection = null;
    this.isConnected = false;
    DatabaseConnection.instance = this;
  }

  static getInstance() {
    if (!this.instance) {
      this.instance = new DatabaseConnection();
    }
    return this.instance;
  }

  connect() {
    if (!this.isConnected) {
      this.connection = "DB Connection Established"
```

# Factory Method Pattern

Das Factory Method Pattern definiert eine Schnittstelle für die Erstellung von Objekten, lässt aber Unterklassen entscheiden, welche Klasse instanziiert werden soll.

- **Flexible Objekterstellung** - Runtime-Entscheidungen
- **Lose Kopplung** - Client kennt konkrete Klassen nicht
- **Erweiterbarkeit** - Neue Produkttypen hinzufügen
- **Single Responsibility** - Erstellung von Verwendung trennen

```
// Abstract Product
class Logger {
  log(message) {
    throw new Error("Must implement log method");
  }
}

// Concrete Products
class ConsoleLogger extends Logger {
  log(message) {
    console.log(`Console: ${message}`);
  }
}

class FileLogger extends Logger {
  log(message) {
    console.log(`File: ${message}`);
  }
}

class DatabaseLogger extends Logger {
  log(message) {
    console.log(`Database: ${message}`);
  }
```

# Observer Pattern

Das Observer Pattern definiert eine one-to-many Abhängigkeit zwischen Objekten, sodass alle Abhängigen automatisch benachrichtigt werden, wenn sich der Zustand des Subjekts ändert.

- **Event-Driven Architecture** - Lose Kopplung zwischen Komponenten
- **Dynamic Relationships** - Observer zur Laufzeit hinzufügen/entfernen
- **Broadcast Communication** - Ein Subjekt, viele Observer
- **Separation of Concerns** - Business Logic von Präsentation trennen

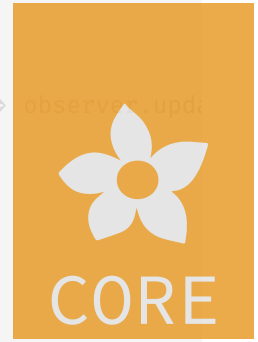
```
// Subject (Observable)
class NewsAgency {
  constructor() {
    this.observers = [];
    this.news = '';
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  unsubscribe(observer) {
    this.observers = this.observers.filter(obs ⇒ ob
  }

  notifyAll() {
    this.observers.forEach(observer ⇒ observer.upd
  }

  setNews(news) {
    this.news = news;
    this.notifyAll();
  }
}
```



# Praktische Übungen

# Übung 1: Singleton Implementation

Implementieren Sie ein Thread-sicheres Singleton Pattern für eine Konfigurationsklasse:

## Anforderungen

- **Thread Safety** - Mehrfacher Zugriff sicher handhaben
- **Lazy Loading** - Instanziierung nur bei Bedarf
- **Configuration Loading** - Einstellungen aus Datei laden
- **Immutable Settings** - Konfiguration nicht veränderbar

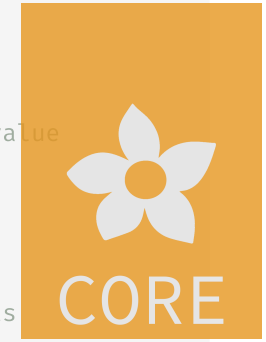
```
// Starter Code
class Configuration {
  constructor() {
    // TODO: Implement singleton logic
  }

  static getInstance() {
    // TODO: Thread-safe singleton creation
  }

  loadSettings(configFile) {
    // TODO: Load configuration from file
  }

  getSetting(key) {
    // TODO: Retrieve configuration value
  }

  // Test helper - only for testing!
  static resetInstance() {
    // TODO: Reset singleton for tests
  }
```



# Übung 2: Factory Method Extension

Erweitern Sie die Logger Factory um neue Logger-Typen:

## Neue Logger-Typen

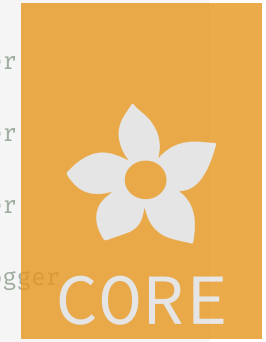
- **EmailLogger** - Send logs via email
- **SlackLogger** - Post to Slack channel
- **MultiLogger** - Combine multiple loggers
- **FilteredLogger** - Log only specific levels
- **AsyncLogger** - Non-blocking logging

## Bonus Challenges

```
// Extend this factory
class LoggerFactory {
  static createLogger(config) {
    const { type, options = {} } = config;

    switch(type) {
      case 'console':
        return new ConsoleLogger(options);
      case 'file':
        return new FileLogger(options);

      // TODO: Add new logger types
      case 'email':
        // TODO: Implement EmailLogger
      case 'slack':
        // TODO: Implement SlackLogger
      case 'multi':
        // TODO: Implement MultiLogger
      case 'filtered':
        // TODO: Implement FilteredLogger
      case 'async':
        // TODO: Implement AsyncLogger
```



# Zusammenfassung & Nächste Schritte

# Key Takeaways

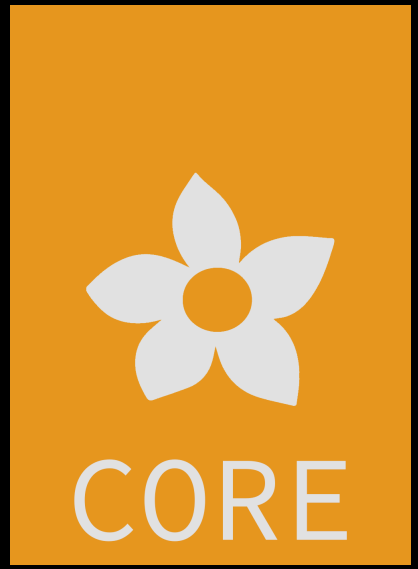
## Patterns Benefits

- **Wiederverwendbare Lösungen** - Bewährte Ansätze für häufige Probleme
- **Kommunikation verbessern** - Gemeinsame Sprache für Entwickler
- **Code-Qualität steigern** - Strukturierte und wartbare Implementierungen
- **Best Practices fördern** - Etablierte Standards und Konventionen
- **Flexibilität erhöhen** - Anpassbare und erweiterbare Architekturen

## Implementation Guidelines

- **Don’t Overengineer** - Patterns nur bei echtem Bedarf einsetzen
- **Context Matters** - Pattern an spezifische Anforderungen anpassen
- **Test Thoroughly** - Pattern-Implementierungen umfassend testen
- **Document Decisions** - Begründung für Pattern-Wahl dokumentieren
- **Team Alignment** - Gemeinsames Verständnis im Team schaffen

# Fragen & Diskussion




**Kontakt:** architecture-training@telekom.de
**Weitere Ressourcen:** Design Patterns Catalog