# Design Patterns

Kennen, erklären und implementieren

# Übersicht

- Was sind Design Patterns?
- Vorteile von Design Patterns
- Kategorien von Design Patterns
- Detaillierte Betrachtung der folgenden Patterns:
  - Singleton
  - Observer
  - Factory
  - Adapter
  - Iterator
  - Strategy
  - Decorator
  - Template Method
  - MVC

# Was sind Design Patterns?

- Bewährte Lösungsansätze für wiederkehrende Problemstellungen in der Softwareentwicklung
- Formalisiert von der "Gang of Four" (GoF) im Buch "Design Patterns: Elements of Reusable Object-Oriented Software" (1994)
- Bieten gemeinsame Sprache für Entwickler
- Fördern besseres, wartbares Code-Design

# Vorteile von Design Patterns

- Wiederverwendbarkeit von bewährten Lösungen
- Verbesserte Kommunikation im Team durch gemeinsame Terminologie
- Erhöhte Code-Qualität und Wartbarkeit
- Verbesserung der Architektur durch Verwendung erprobter Strukturen
- Leichtere Erweiterbarkeit des Systems

# Kategorien von Design Patterns

- Erzeugungsmuster (Creational Patterns)
  - Singleton, Factory
- Strukturmuster (Structural Patterns)
  - Adapter, Decorator
- Verhaltensmuster (Behavioral Patterns)
  - Observer, Iterator, Strategy, Template Method
- Architekturmuster (Architectural Patterns)
  - MVC, Registry

# Singleton Pattern

- Zweck: Sicherstellen, dass eine Klasse nur eine Instanz hat und globalen Zugriff darauf bietet.
- Typische Anwendungsfälle:
  - Datenbankverbindungen
  - Logger
  - Konfigurationsmanager

```
public class Singleton {
    private static Singleton instance;

    private Singleton() { }

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

https://www.youtube.com/watch?v=zRdpK8q7iYM

# Observer Pattern

Zweck: Definiert eine abhängigkeit zwischen Objekten, sodass bei Zustandsänderung eines Objekts alle abhängigen Objekte benachrichtigt werden.

- Typische Anwendungsfälle:
  - Event-Handling-Systeme
  - MVC-Implementierungen (Model benachrichtigt Views)
  - Publish-Subscribe-Architekturen

https://www.youtube.com/watch?v=bLQVsvEGQFU

```
// Subject-Interface
interface Subject {
   void registerObserver(Observer o);
   void removeObserver(Observer o);
   void notifyObservers();
}

// Observer-Interface
interface Observer {
   void update(String message);
}
```

# Factory Pattern(Die Fabrikmethode)

Zweck: Erstellt Objekte ohne ihre konkrete Klasse zu spezifizieren.

- Typische Anwendungsfälle:
  - Framework-Implementierungen
  - Produktfamilien mit gemeinsamen Interfaces
  - Wenn die Erstellung eines Objekts komplex ist oder Wissen über Implementierungsdetails erfordert

https://www.youtube.com/watch?v=WQTIgChLdhQ

# Adapter Pattern

Zweck: Konvertiert die Schnittstelle einer Klasse in eine andere Schnittstelle, die Clients erwarten.

- Typische Anwendungsfälle:
  - Integration von Drittanbieter-Bibliotheken
  - Legacy-Code-Integration
  - Wenn verschiedene Systeme miteinander kommunizieren müssen

https://www.youtube.com/watch?v=VKbbVmBpfpQ

```
// Target interface
interface Target {
    void request();
}

// Adaptee (incompatible interface)
class Adaptee {
    void specificRequest() {
        System.out.println("Specific request");
    }
}

// Adapter
class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}
```

# Iterator Pattern

**Zweck:** Bietet eine Möglichkeit, auf Elemente einer Sammlung zuzugreifen, ohne die zugrunde liegende Struktur offenzulegen.

- **Typische Anwendungsfälle:**
  - Durchlaufen von komplexen Datenstrukturen (Bäume, Graphen)
  - Wenn verschiedene Traversierungsalgorithmen benötigt werden
  - Einheitliche Schnittstelle für verschiedene Sammlungstypen

https://www.youtube.com/watch?v=HAodzjQ-pSI

```
interface Iterator<T> {
    boolean hasNext();
    T next();
}

class ConcreteCollection<T> {
    private List<T> items = new ArrayList<>();

    public void add(T item) {
        items.add(item);
    }

    public Iterator<T> createIterator() {
        return new ConcreteIterator<>(items);
    }
}
```

# Strategy Pattern

**Zweck:** Definiert eine Familie von Algorithmen, kapselt jeden einzelnen und macht sie austauschbar.

- **Typische Anwendungsfälle:**
  - Verschiedene Algorithmen zur Laufzeit auswählen
  - Vermeidung komplexer Bedingungslogik
  - Wenn ein Algorithmus variieren soll, ohne Client-Code zu ändern

https://www.youtube.com/watch?v=5VlBrlFJ0kU

```
// Strategy interface
interface SortStrategy {
    void sort(int[] array);
}

// Context
class Sorter {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}
```

# Decorator Pattern

**Zweck:** Fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu.

- **Typische Anwendungsfälle:**
  - Erweiterung der Funktionalität zur Laufzeit
  - Stapelbare Funktionalität
  - Alternative zur Unterklassenbildung

https://www.youtube.com/watch?v=wA344k4tPz8

```
// Component interface
interface Component {
    String operation();
}

// Concrete Component
class ConcreteComponent implements Component {
    @Override
    public String operation() {
        return "Basic operation";
    }
}

// Decorator
abstract class Decorator implements Component {
    protected Component component;

    public Decorator(Component component) {
        this.component = component;
    }
}

// Concrete Decorator
class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component component) {
        super(component);
    }

    @Override
    public String operation() {
        return "Decorated(" + component.operation() + ")";
    }
}
```

# Template Method Pattern

**Zweck:** Definiert das Skelett eines Algorithmus in einer Operation und überlässt einige Schritte den Unterklassen.

- **Typische Anwendungsfälle:**
  - Framework-Design
  - Variationen eines komplexen Algorithmus
  - Wenn gemeinsamer Code in mehreren Klassen vorhanden ist

https://www.youtube.com/watch?v=BdJR55WHaqw

```
abstract class AbstractClass {
    // Template method
    public final void templateMethod() {
        step1();
        step2();
        hook();
        step3();
    }

    protected abstract void step1();
    protected abstract void step2();

    protected void step3() {
        System.out.println("Default step 3");
    }

    // Hook - kann überschrieben werden
    protected void hook() { }
}
```

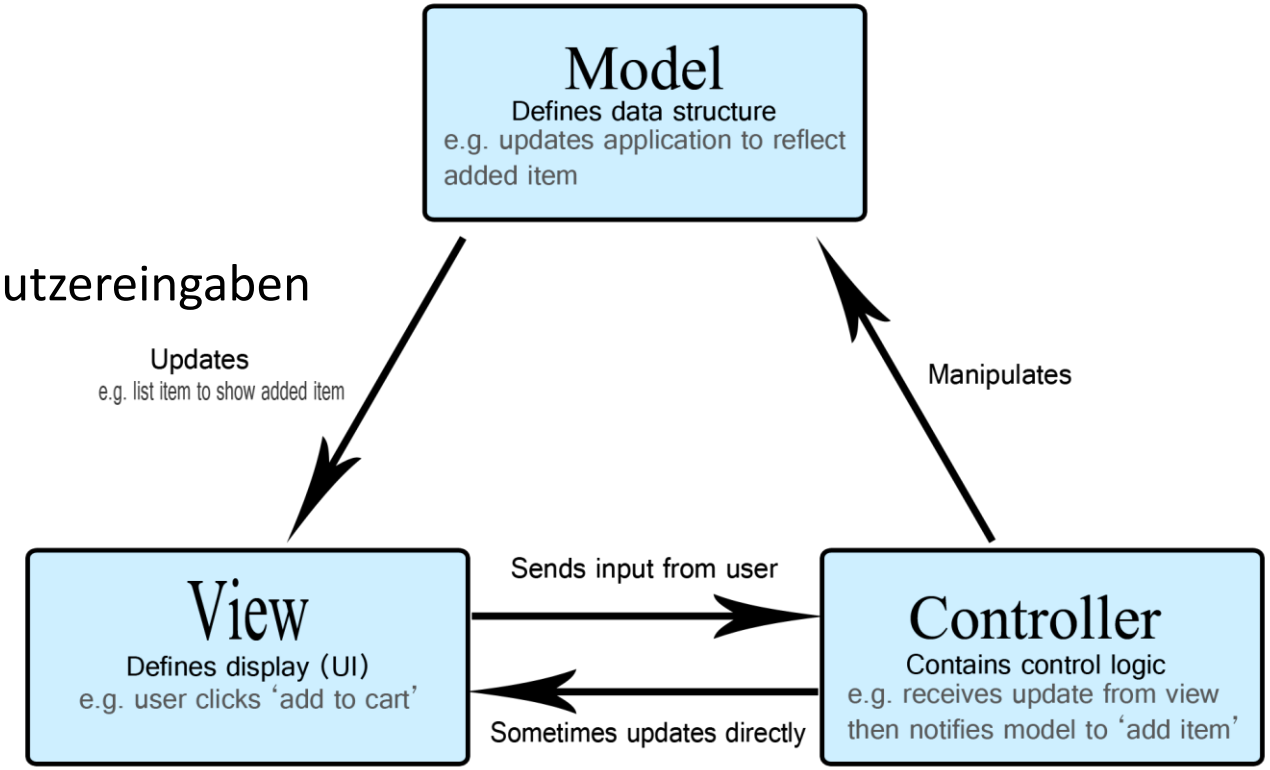
# MVC Pattern (Model-View-Controller)

**Zweck:** Trennt eine Anwendung in drei Hauptkomponenten: Model, View und Controller.

- **Typische Anwendungsfälle:**
  - Web-Anwendungen
  - Desktop-Anwendungen mit Benutzeroberfläche
  - Komplexe Benutzerschnittstellen
- **Model:** Daten und Geschäftslogik
- **View:** Präsentation/UI
- **Controller:** Verbindet Model und View, verarbeitet Benutzereingaben

https://www.youtube.com/watch?v=6oRPgM_ZxUY

https://www.youtube.com/watch?v=6m9Vxz3An0Y



# Best Practices für Design Patterns

- Pattern nicht um des Patterns willen verwenden
- Pattern an Kontext und Anforderungen anpassen
- Kombination von Patterns für komplexe Probleme
- Dokumentation der verwendeten Patterns zur Kommunikation
- Berücksichtigung der Performance-Implikationen

# Fazit & Diskussion

- Design Patterns sind wichtige Werkzeuge im Software-Entwicklungs-Prozess
- Sie bieten bewährte Lösungen für häufige Probleme
- Kenntnis der Pattern-Sprache verbessert Kommunikation im Team
- Wichtig: Das richtige Pattern für das richtige Problem wählen
- Fragen?