Vortrag über das „Strategy-Pattern“ bei Design-Pattern von David Ehrenberg FI 37 Anwender

**Einleitung (2-3 Minuten)**

1. **Begrüßung und Einführung:**

Hallo und herzlich willkommen zu meinem Vortrag über das **Strategy Pattern** bei Design Patterns in der Softwareentwicklung.

1. **Was sind Design Patterns?**
   * Definition:

Design Patterns sind bewährte Lösungen für wiederkehrende Probleme in der Softwareentwicklung. Sie bieten eine standardisierte Herangehensweise zur Lösung bestimmter Aufgaben und helfen, flexibleren und wartbaren Code zu schreiben.

1. **Was ist das Strategy Pattern?**
   * Definition:

Das Strategy Pattern ist ein Verhaltensmuster, das es uns ermöglicht, eine Familie von Algorithmen zu definieren und diese zur Laufzeit auszuwählen, ohne dass der Client – also der Code, der die Algorithmen verwendet – von deren konkreter Implementierung abhängt.

**Warum brauchen wir das Strategy Pattern? (3-4 Minuten)**

1. **Problemstellung ohne das Strategy Pattern:**
   * Typisches Beispiel:

Stellet euch vor, wir entwickeln eine Anwendung, die verschiedene Berechnungslogiken oder Verhaltensweisen, wie z.B. Zahlungsarten oder Sortieralgorithmen, verwenden soll. Ohne das Strategy Pattern müssten wir oft mit vielen if-else oder switch-Anweisungen arbeiten, um das passende Verhalten auszuwählen.

1. **Problem mit if-else Statements:**
   * warum if-else und switch-Anweisungen problematisch sind:

Diese Art von Code kann schnell unübersichtlich und schwer wartbar werden, besonders wenn wir immer wieder neue Logiken hinzufügen müssen. Jede Änderung an den Algorithmen oder neuen Varianten führt dazu, dass wir den Code an mehreren Stellen anpassen müssen – was gegen das Prinzip der Wartbarkeit verstößt.

1. **Einführung in das Strategy Pattern als Lösung:**
   * Wie löst das Strategy Pattern dieses Problem:

Das Strategy Pattern löst dieses Problem, indem es die Auswahl des Algorithmus oder Verhaltens aus dem Code herausnimmt und stattdessen in eigene Klassen kapselt. Dadurch wird der Code modularer, besser wartbar und kann leicht erweitert werden.

**Aufbau des Strategy Patterns (5 Minuten)**

1. **Komponenten des Strategy Patterns:**
   * die wichtigsten Bestandteile des Strategy Patterns:

Das Strategy Pattern besteht aus drei Hauptkomponenten:

* + - **Strategy Interface**: Ein Interface, das den gemeinsamen Vertrag für die Algorithmen definiert.
    - **Concrete Strategies**: Konkrete Implementierungen dieses Interfaces, die verschiedene Algorithmen oder Verhaltensweisen enthalten.
    - **Context**: Die Klasse, die das Strategy Interface verwendet und entscheidet, welche konkrete Strategie zur Laufzeit genutzt wird.

1. **UML-Diagramm (optional):**

Ein UML-Diagramm des Strategy Patterns zeigt, wie der Context die Strategy verwendet, ohne von den konkreten Strategien abhängig zu sein. Die konkreten Strategien implementieren das Strategy-Interface, wodurch sie austauschbar sind.

Hier ist ein einfaches UML-Diagramm zur Veranschaulichung:

+----------------+ +----------------+

| Context | | Strategy |

|----------------| |----------------|

| - strategy: |<>------->| + algorithm() |

| Strategy | +----------------+

|----------------| | |

| + execute() | | |

+----------------+ +----------------+

| ^

| |

v |

+-------------------+ +------------------+

| ConcreteStrategyA | |ConcreteStrategyB |

|-------------------| |------------------|

| + algorithm() | | + algorithm() |

+-------------------+ +------------------+

1. **Kurze Übersicht über das Open-Closed-Prinzip:**
   * Strategy Pattern unterstützt das **Open-Closed-Prinzip** aus den SOLID-Prinzipien

Das Open-Closed-Prinzip besagt, dass eine Klasse offen für Erweiterungen, aber geschlossen für Änderungen sein sollte. Mit dem Strategy Pattern können wir neue Verhaltensweisen hinzufügen, ohne den bestehenden Code zu verändern.

**Beispiel eines Problems und Lösung mit dem Strategy Pattern (7-8 Minuten)**

1. **Das Problem: Zahlungsarten im Online-Shop:**
   * ein konkretes Szenario

Nehmen wir an, wir entwickeln eine E-Commerce-Anwendung, in der ein Kunde auf verschiedene Arten bezahlen kann – etwa mit Kreditkarte, PayPal oder bar. Jede dieser Bezahlmethoden hat eine andere Logik.

1. **Ohne das Strategy Pattern:**

Ohne das Strategy Pattern würden wir wahrscheinlich viele if-else-Anweisungen in der ShoppingCart-Klasse haben, um die richtige Bezahlmethode zu wählen. Zum Beispiel:

*if (paymentMethod.equals("CreditCard")) {*

*// Kreditkartenzahlung*

*} else if (paymentMethod.equals("PayPal")) {*

*// PayPal-Zahlung*

*} else if (paymentMethod.equals("Cash")) {*

*// Barzahlung*

*}*

1. **Lösung mit dem Strategy Pattern (Kurzversion):**

Mit dem Strategy Pattern können wir die Logik in separate Klassen auslagern. Wir erstellen ein gemeinsames PaymentStrategy-Interface, das jede Zahlungsmethode implementiert. Der Warenkorb (der Context) muss nun nicht mehr wissen, wie jede Zahlungsart funktioniert – er delegiert die Bezahlung einfach an die ausgewählte Strategie.

1. **Codebeispiel:**

***// Strategy Interface***

public interface PaymentStrategy {

void pay(int amount);

}

***// Concrete Strategies***

public class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Bezahlter Betrag mit Kreditkarte: " + amount);

}

}

public class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Bezahlter Betrag mit PayPal: " + amount);

}

}

***// Context***

public class ShoppingCart {

private PaymentStrategy paymentStrategy;

public ShoppingCart(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(int amount) {

paymentStrategy.pay(amount);

}

}

1. **Vorteile der Lösung:**

Diese Lösung ist viel flexibler, weil:

* + - Wir können neue Zahlungsarten hinzufügen, ohne den Code des Warenkorbs ändern zu müssen.
    - Die Bezahlstrategien sind leicht testbar, weil sie voneinander unabhängig sind.
    - Der Code ist modular und leichter zu pflegen.

**Vorteile und Nachteile des Strategy Patterns (3 Minuten)**

1. **Vorteile:**
   * **Flexibilität und Erweiterbarkeit**: Neue Algorithmen oder Verhaltensweisen können einfach hinzugefügt werden, ohne bestehenden Code zu ändern.
   * **Saubere Trennung der Verantwortlichkeiten**: Jedes Verhalten ist in einer eigenen Klasse kapselt.
   * **Einfaches Testen**: Jede Strategie kann separat getestet werden.
2. **Nachteile:**
   * **Komplexität**: Wenn zu viele Strategien entstehen, kann dies zu einer großen Anzahl von Klassen führen.
   * **Verwaltungsaufwand**: Der Client muss wissen, welche Strategie er verwenden will, was zu mehr Verwaltungsaufwand führen kann.

**Fazit (2 Minuten)**

1. **Zusammenfassung:**
   * Wiederhole Hauptpunkte:

Das Strategy Pattern ist ein wertvolles Werkzeug, um wiederkehrende Probleme in der Softwareentwicklung elegant zu lösen. Es trennt Algorithmen oder Verhaltensweisen in eigene Klassen und sorgt so für Flexibilität, Erweiterbarkeit und sauberen Code.

1. **Wichtiger Hinweis:**
   * **Wann sollte das Strategy Pattern verwendet werden?**

Das Strategy Pattern ist besonders nützlich, wenn sich bestimmte Verhaltensweisen häufig ändern oder erweitert werden müssen, und der Code sauber und flexibel bleiben soll.

1. **Schlusswort:**

Vielen Dank für Eure Aufmerksamkeit. Ich hoffe, der Vortrag konnte Euch das Strategy Pattern näherbringen.

„Ohne das Strategy Pattern würden wir wahrscheinlich viele if-else-Anweisungen in der ShoppingCart-Klasse haben, um die richtige Bezahlmethode zu wählen.

**Codebeispiel:**

*if (paymentMethod.equals("CreditCard")) {*

*// Kreditkartenzahlung*

*} else if (paymentMethod.equals("PayPal")) {*

*// PayPal-Zahlung*

*} else if (paymentMethod.equals("Cash")) {*

*// Barzahlung*

*}*

Mit dem Strategy Pattern können wir die Logik in separate Klassen auslagern. Wir erstellen ein gemeinsames PaymentStrategy-Interface, das jede Zahlungsmethode implementiert. Der Warenkorb (der Context) muss nun nicht mehr wissen, wie jede Zahlungsart funktioniert – er delegiert die Bezahlung einfach an die ausgewählte Strategie.“

**Codebeispiel:**

***// Strategy Interface***

public interface PaymentStrategy {

void pay(int amount);

}

***// Concrete Strategies***

public class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Bezahlter Betrag mit Kreditkarte: " + amount);

}

}

public class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Bezahlter Betrag mit PayPal: " + amount);

}

}

***// Context***

public class ShoppingCart {

private PaymentStrategy paymentStrategy;

public ShoppingCart(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(int amount) {

paymentStrategy.pay(amount);

}

}