Bachelor Seminar: Design Patterns

Strategy Design Pattern

Aimad Bouchouaf

Bergische Universität Wuppertal

14.01.2025

Inhalt

Inhalt

Definition

- Das Strategie-Muster ist ein **Verhaltensmuster**
- Verhaltensmuster ermöglichen eine effiziente Aufgabenverteilung, ohne die Systemstruktur zu verändern.

- Das Strategie-Muster ermöglicht den **Austausch verschiedener Algorithmen** oder Verhaltensweisen, **ohne den Code** des Objekts zu **ändern**.
- Es regelt die **Interaktion** und **Kommunikation** zwischen Objekten.

Inhalt

Klassendiagramm

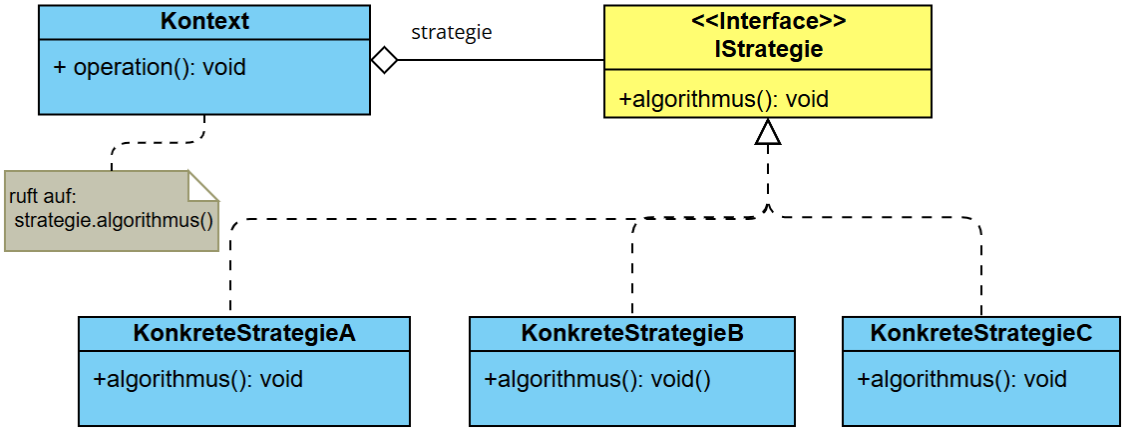


Abbildung: Klassendiagramm des Strategiemusters

- **Kontext:**
  - Aggregiert die Schnittstelle `IStrategie`.
  - Enthält eine Referenz `strategie` vom Typ `IStrategie`.
  - Ruft die Methode `strategie.algorithmus()` auf, um `operation()` auszuführen.

- **IStrategie:**
  - Schnittstelle, die von allen unterstützenden Algorithmen implementiert wird.
  - Ermöglicht den Aufruf von Algorithmen, die in `KonkreteStrategieX` definiert sind.

- **KonkreteStrategieX (X = A..Z):**
  - Implementiert einen spezifischen Algorithmus.
  - Der Methodenkopf wird in der Schnittstelle `IStrategie` deklariert.

Interaktion

- **Zusammenarbeit zwischen Strategy und Kontext**:
  - Der Kontext nutzt die Strategie, um den gewählten Algorithmus auszuführen.
  - Der Kontext kann Daten an die Strategie übergeben oder sich selbst bereitstellen, damit die Strategie darauf zugreifen kann.

- **Weiterleitung von Anfragen**:
  - Der Kontext leitet **Anfragen vom Client an die Strategie** weiter.
  - Der Client **erstellt eine konkrete Strategie** und übergibt sie dem Kontext. Danach interagiert der Client nur mit dem **Kontext**.

- **Auswahl von Strategien**:
  - Es gibt **mehrere konkrete Strategien**, aus denen der **Client** die passende **auswählen** kann.

Sequenzdiagramm Strategie

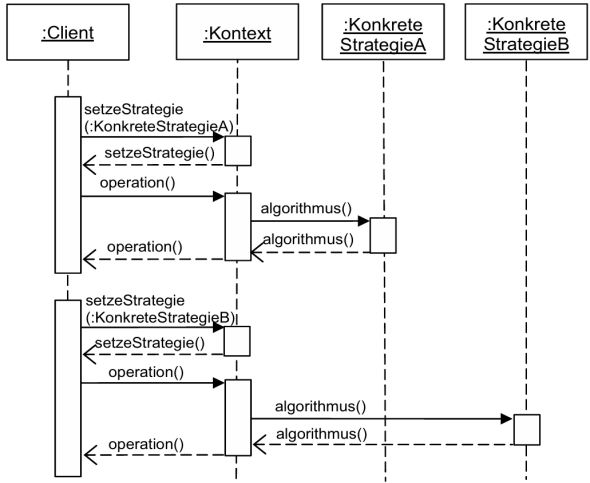
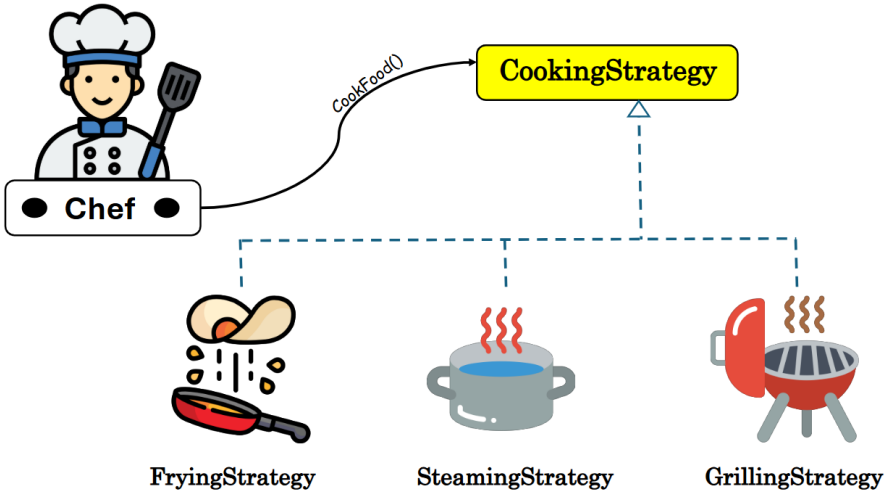


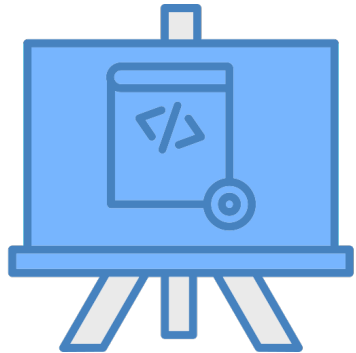
Abbildung: Sequenzdiagramm Strategie

Analogie aus dem Alltag



Inhalt

Strategie im Einsatz



Spring Security Authentication Strategies

- Spring Security nutzt das **Strategy Design Pattern**, um unterschiedliche **Authentifizierungsstrategien** zu ermöglichen.
- Beispiel: Verschiedene Authentifizierungsmechanismen wie **Formular-Login** und **HTTP Basic Authentication** können als Strategien bereitgestellt werden.

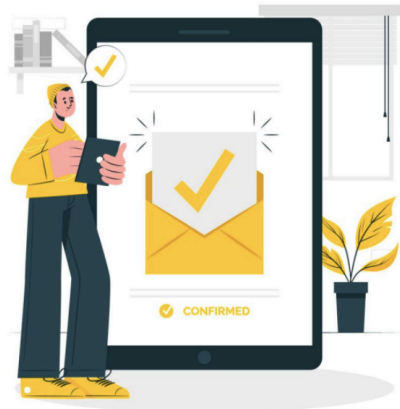
Spring Security Authentication Strategies

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .httpBasic();
    }
}
```

Inhalt

Was ist das Problem ?

- Entwicklung einer E-Commerce-Plattform mit **kundengruppenspezifischen Preisen**:
  - Preise variieren je nach Mitgliedschaft **Regulär, Gold, Premium** .
- **Verschiedene Zahlungsoptionen** für Kunden:
  - **PayPal, Visa-Karte, Banküberweisung** .



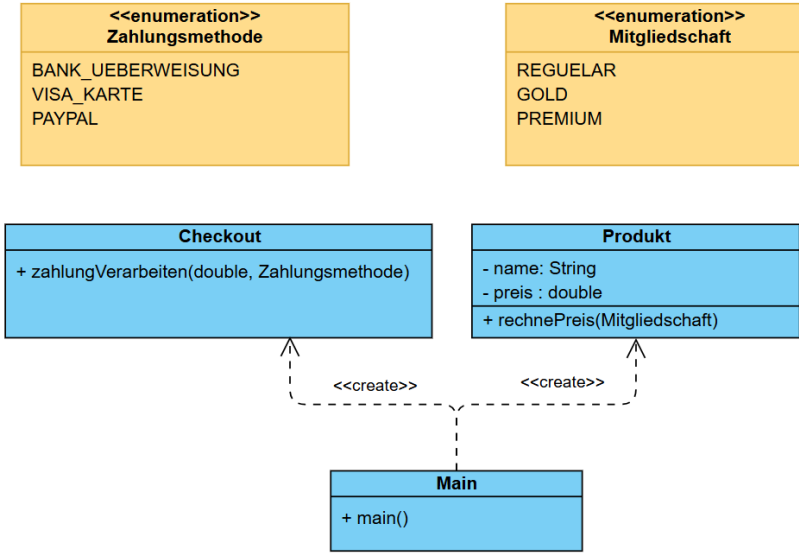


Abbildung: Naive Lösung

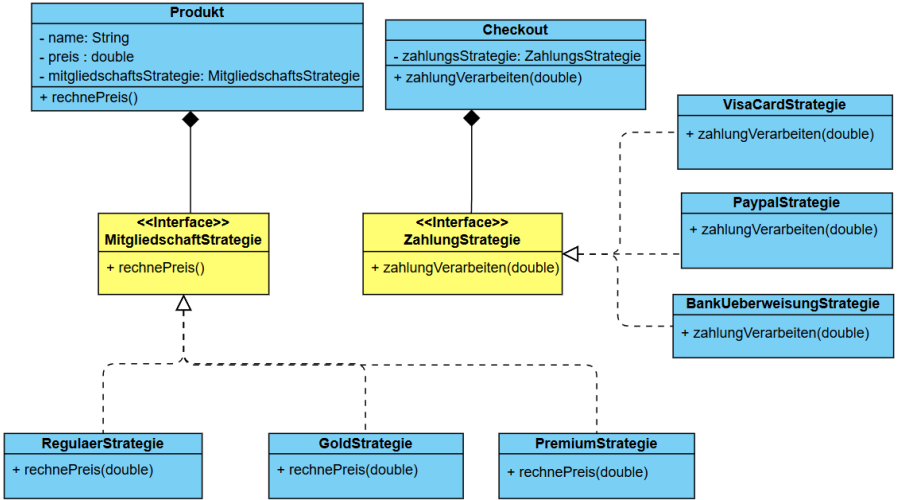


Abbildung: Optimierte Lösung

Inhalt

Vorteile

- **Flexibilität zur Laufzeit**:
  Algorithmen können zur **Laufzeit** ausgetauscht werden, **ohne den Code zu ändern**.
- **Unabhängigkeit vom Kontext**:
  Der Kontext ist nur von der **Schnittstelle**, nicht von der konkreten Implementierung **abhängig**.
- **Vermeidung von Fallunterscheidungen**:
  Keine komplexen Verzweigungen im Code, jede Entscheidung wird in einer **eigenen Strategieklasse** getroffen.

Nachteile

- **Applikation muss Strategien kennen**:
  Eine Applikation bzw. die Umgebung des Kontextobjekts muss wissen, **welche** Strategie **in welcher** Situation verwendet wird, um das Kontextobjekt richtig zu konfigurieren.
- **Viele Klassen**:
  Es werden viele, **oft kleine** Klassen geschrieben.
- **Erhöhung des Kommunikationsaufwands**:
  Die gemeinsame Schnittstelle für alle Strategien kann **unnötig viele** Parameter erfordern, was den Kommunikationsaufwand zwischen Kontext und Strategie erhöht.
- **Verwaltung der Strategien**:
  - Funktionstypen für Algorithmus-Versionen.
  - Anonyme Funktionen statt Strategieobjekte.
  - Weniger Klassen und Schnittstellen.

Beispiel

```
import { useState } from 'react';
const addStrategy = (a, b) => a + b;
const subtractStrategy = (a, b) => a - b;
const multiplyStrategy = (a, b) => a * b;
function App() {
  const [strategy, setStrategy] = useState(()=> addStrategy);
  const [result, setResult] = useState(0);

  const handleCalculation = (a, b) => {
  setResult(strategy(a, b));};
  return (
  <div> <h1>Strategy Pattern in React</h1>
  <button onClick={() => setStrategy(() =>addStrategy)}>Add</button>
  <button onClick={() => setStrategy(() =>subtractStrategy)}>Subtract</button>
  <button onClick={() => setStrategy(() =>multiplyStrategy)}>Multiply</button>
    <div>
    <button onClick={() => handleCalculation(5, 3)}>Calculate 5 and 3</button>
    </div>
    <div>Result: {result}</div>
  </div>
  )};
```

Strategie-Anwendbarkeit

1. **Dynamischer Algorithmuswechsel**: Das Strategie-Muster ist ideal, wenn man mehrere Algorithmen hat und zur Laufzeit flexibel zwischen ihnen wechseln möchte.

2. **Unterschiedliche Implementierungen für dieselbe Aufgabe**: Wenn verschiedene Klassen **dieselbe Operation auf unterschiedliche Weise** ausführen, eignet sich das Strategie-Muster, um diese Variationen sauber zu handhaben.

3. **Vermeidung komplexer Bedingungen**: Bei vielen bedingten Anweisungen hilft das Strategie-Muster, den Code zu vereinfachen, indem es die **Verzweigungen** durch Polymorphismus ersetzt.

Zusammenfassung

- Das **Strategie-Muster** ermöglicht die **dynamische Auswahl von Algorithmen** zur Laufzeit, ohne den Code der beteiligten Objekte zu verändern.
- Es fördert **Flexibilität**, indem es verschiedene Strategien **austauschbar** macht und so die **Interaktion zwischen Objekten optimiert**.
- Durch das **Entkoppeln von Strategie und Kontext** wird die **Erweiterbarkeit des Systems** erheblich verbessert.

Quellen

- **Architektur- und Entwurfsmuster der Softwaretechnik**, J. Goll, M. Koller, M. Watzko.
- **Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software**, E. Gamma, R. Helm, R. Johnson, J. Vlissides.
- `https://refactoring.guru/design-patterns/strategy`.
- `https://www.udemy.com/ultimate-design-patterns/`.
- `https://www.flaticon.com/` (Bilder).

Vielen Dank für Eure Aufmerksamkeit!