

# PROGRAMMENTWURF - PROTOKOLL

zu



Robin Schwenzfeier - 4868455 - TINF22B5

# **KAPITEL 1: EINLEITUNG (4P)**

# *INHALT DES KAPITELS*

## **1. ÜBERSICHT ÜBER DIE APPLIKATION (1P)**

- Name und Bedeutung
- Features und Funktionen
- Probleme, die sie löst

## **2. STARTEN DER APPLIKATION (1P)**

- Voraussetzungen
- Schritte zum Starten
- Alternative zum Starten

## **3. TECHNISCHER ÜBERBLICK (2P)**

- Java (JDK 17)
- Maven
- JSON & GSON
- JUnit & JaCoCo
- Clean Architecture

# ÜBERSICHT ÜBER DIE APPLIKATION (1P)

*Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?*

# **ULRICA: NAME ERKLÄRT**

*Ein Acronym.*

**U**niversa**L** **R**ange and dest**I**nation **C**Alculator

# FEATURES DER APPLIKATION

*Probleme, die sie löst.*

- 1. Erstellt und verwaltet Fahrzeugprofile für Elektrofahrzeuge
- 2. Berechnet verbleibende Reichweite von Elektrofahrzeugen
- 3. + 4. Berechnet Ladezeiten an DC- und AC-Ladestationen
- 5. Berechnet und simmulierte eine Route

# FEATURES DER APPLIKATION

## 1. FAHRZEUGPROFIL-MANAGEMENT

- Erstellung, Anzeige und Löschung von Fahrzeugprofilen
- Konfiguration von Batteriedaten (Typ, Kapazität, Degradation, etc.)
- Definition von Verbrauchsprofilen (bei 50km/h, 100km/h, 130km/h)

## 2. REICHWEITENBERECHNUNG

- Strategie-Pattern für verschiedene Berechnungsmethoden
- WLTP-basierte Berechnung mit "echten" Bedingungen
- Temperaturinflüsse
- Geländebedingungen
- etc.

# FEATURES DER APPLIKATION

## 3. DC (SCHNELLES) LADEN

- SoC-basierte Berechnungen
- Temperatureinflüsse beim Laden
- Leistungsreduktion basierend auf Batteriezustand
- Detaillierte Ladezeitschätzungen
- Berücksichtigung von Batterie-Typ

## 4. AC (LANGSAMES) LADEN

- Verschiedene Anschlusstypen (Haushalt, Camping, Wallbox)
- Berechnung von Effizienzverlusten
- Temperatur-Effizienzfaktoren
- Ladezeitprognosen

# **STARTEN DER APPLIKATION (1P)**

*Wie startet man die Applikation? Was für Voraussetzungen werden benötigt?  
Schritt-für-Schritt-Anleitung*

# VORAUSSETZUNGEN:

- Java 17 oder höher
- Maven
- Git

# SCHRITTE ZUM STARTEN:

1. Repository klonen: `git clone https://github.com/mausio/ULRICA`
2. In Projektverzeichnis wechseln: `cd /path/to/ULRICA`
3. Projekt kompilieren: `mvn clean compile`
4. ULRICA starten: `mvn exec:java -Dexec.mainClass="org.ulrica.App"`

# ALTERNATIVE ZUM STARTEN:

1. JAR-Datei erstellen: `mvn clean package`

- Dieser Befehl kompiliert den Code und erstellt eine ausführbare JAR-Datei im Verzeichnis target/
- Die JAR-Datei wird als `ULRICA-1.0-SNAPSHOT.jar` gespeichert

2. JAR-Datei ausführen: `java -jar target/ULRICA-1.0-SNAPSHOT.jar`

- Stellt sicher, dass Java 17 oder höher installiert ist
- Die JAR-Datei enthält alle notwendigen Abhängigkeiten
- Kann auf jedem System mit Java 17+ ausgeführt werden

# TECHNISCHER ÜBERBLICK (2P)

*Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils  
Begründung für den Einsatz der Technologien*

# TECHNOLOGIEN: ÜBERBLICK

- **Java (JDK 17):** Objektorientierte Programmiersprache
- **Maven:** Build-Management-Tool
- **JSON/GSON:** Datenaustauschformat und Java-Bibliothek für Persistenz
- **JUnit 4:** Test-Framework für automatisierten Tests
- **JaCoCo:** Code-Coverage-Tool
- **Github Workflow:** CI/CD Pipeline-Tool mit Maven Build

# JAVA (JDK 17)

## VORTEILE:

- Plattformunabhängigkeit durch Java Virtual Machine
- Starke objektorientierte Programmierung
- Typ-Sicherheit bei Laufzeit
- LongTermSupport
- Umfangreiche Bibliotheken verfügbar -> GSON
- Ideale Grundlage für Clean Architecture und Implementierung von Prinzipien

## ALTERNATIVEN:

- **Kotlin:**  
Eleganter, aber weniger etabliert (auch nicht erlaubt)
- **Python:**  
Einfacher, aber weniger performant und relativ unbekannt für mich
- **JavaScript:**  
Mir sehr bekannt, aber schwierig Architekturprinzipien zu implementieren (auch nicht erlaubt)

# MAVEN

## VORTEILE

- "Plug&Play" für Nutzer:innen
- Konsistente Projektstruktur
- Automatisiertes Abhängigkeitsmanagement
- Standardisierte Build-Lebenszyklen
- Integration mit JUnit und JaCoCo
- Plugin-Ökosystem für erweiterte Funktionen

## KONFIGURATION IN ULRICA

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.10.1</version>
  </dependency>
</dependencies>
```

# JSON & GSON

## VORTEILE

- Leichtgewichtiges Datenformat
- Für Menschen lesbar und einfach zu bearbeiten
- Ideal für Konfigurationsdaten, e.g. Fahrzeugprofile
- Weit verbreitet und unterstützt
- GSON bietet einfache Java-Integration
- TypeAdapter für komplexe Datentypen

## IMPLEMENTIERUNG

```
public class JsonCarProfileRepository  
    implements CarProfilePersistencePortInterface { //  
    private final Gson gson; // Gson-Instanz für JSON  
    private final Path filePath; // Pfad zur JSON-Da  
  
    public JsonCarProfileRepository() {  
        this.gson = new GsonBuilder()  
            .setPrettyPrinting() // Formatiert die JSON-A  
            .registerTypeAdapterFactory( // Ermöglicht d  
                new OptionalTypeAdapterFactory()) // Regis  
            .create(); // Factory: Entwurfsmuster (Design  
        this.filePath = Paths.get(  
            STORAGE_DIR, FILE_NAME); // Definiert den Spe  
    }
```

# TEST-FRAMEWORK: JUNIT

## VORTEILE

- De-facto Standard für Java-Tests
- Umfangreiche Assertions-Bibliothek
- Integration mit Build-Tools und JaCoCo
- Test-getriebene Entwicklung (TDD) wird unterstützt

# CODE-COVERAGE: JACOCO

*Fokus auf qualitative Tests u. kritische Komponenten, statt bloßer Maximierung*

## VORTEILE

- Detaillierte Abdeckungsmetriken
- Integration mit Maven und CI/CD
- Identifikation von ungetesteten Codebereichen
- Visualisierung der Testabdeckung
- Qualitätssicherung während der Entwicklung
- Motivation zur Erhöhung der Testabdeckung

## ERGEBNISSE

- Gesamtdeckung: 47%
- Domain Layer: 68%
- Application Layer: 53%
- Presentation Layer: 42%
- Infrastructure Layer: 36%
- Core-Komponenten: 75%

# GITHUB WORKFLOW

*Kontinuierliche Integration und automatische Qualitätssicherung*

## VORTEILE

- Automatisierte Builds bei jedem Commit
- Kontinuierliche Ausführung der Tests
- Automatische Code-Coverage-Analyse
- Codequalitätsmessung mit CLOC
- E-Mail-Benachrichtigungen über Build-Status
- Früherkennung von Fehlern und Qualitätsproblemen

## IMPLEMENTIERUNG

```
name: Java CI with Maven

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
```

# **KAPITEL 2: SOFTWAREARCHITEKTUR (8P)**

# *INHALT DES KAPITELS*

## 1. GEWÄHLTE ARCHITEKTUR (4P)

- Clean Architecture
- Architekturschichten
- Analyse und Begründung
- UML der wichtigsten Klassen

## 2. DOMAIN CODE (1P)

- Definition von Domain Code
- Code-Beispiel aus ULRICA
- Analyse der Domain-Schicht

## 3. ANALYSE DER DEPENDENCY RULE (3P)

- Positives Beispiel
- Negatives Beispiel
- UML der Abhängigkeiten
- Lösungsansätze

# **GEWÄHLTE ARCHITEKTUR (4P)**

*In der Vorlesung wurden Softwarearchitekturen vorgestellt. Welche Architektur wurde davon umgesetzt? Analyse und Begründung inkl. UML der wichtigsten Klassen, sowie Einordnung dieser Klassen in die gewählte Architektur*

# CLEAN ARCHITECTURE

- **Application Layer:**

Implementiert Use Cases (Interfaces) über Interaktoren und Ports für Kommunikation

- **Domain Layer:**

Enthält die Entität, Value Objects, Services (z.B. Kalkulation) und Application State

- **Infrastructure Layer:**

Konkrete Implementierungen von Repositories und Adapters und Utils

- **Presentation Layer:**

Steuert die Benutzerinteraktion über Controller und Views und Utils

```
→ ulrica git:(main) ✘ tree -d
```

```
.
```

```
  └── application
```

```
    ├── port
```

```
    │   ├── in
```

```
    │   └── out
```

```
    └── service
```

```
        └── usecase
```

```
── domain
```

```
    ├── entity
```

```
    └── service
```

```
        └── valueobject
```

```
── infrastructure
```

```
    ├── adapter
```

```
    └── persistence
```

```
        └── util
```

```
── presentation
```

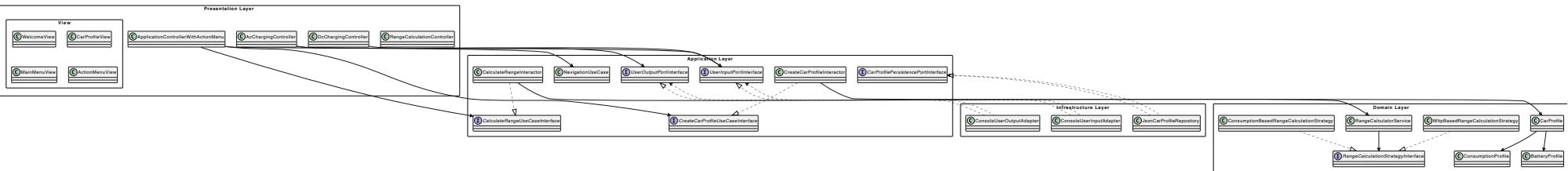
```
    ├── controller
```

```
    └── util
```

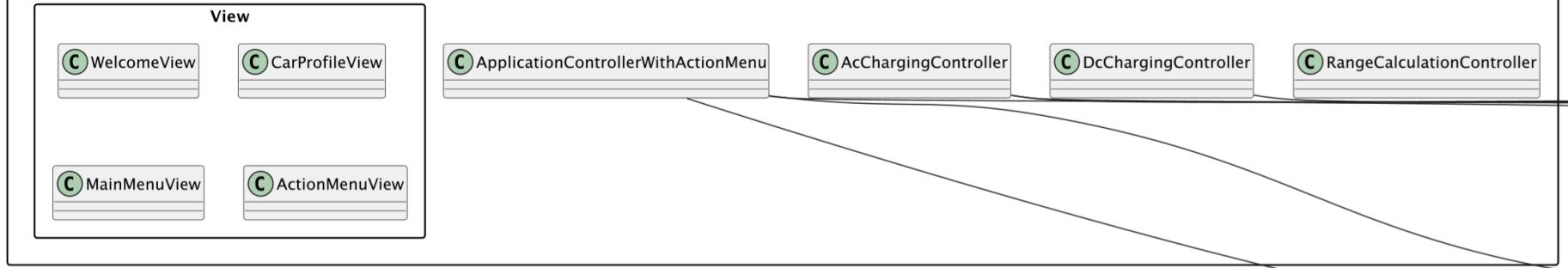
```
        └── view
```

# CLEAN ARCHITECTURE: UML

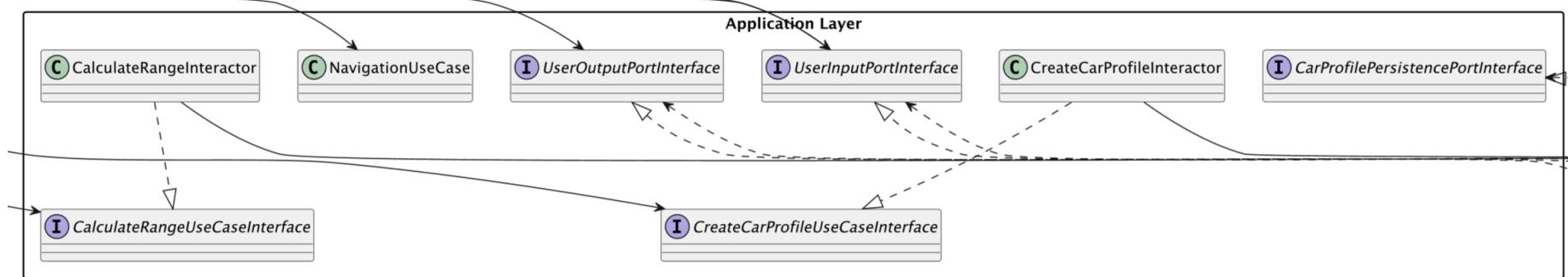
- Domänenlogik ist unabhängig von externen Frameworks/Bibliotheken
- Application Layer enthält anwendungsspezifische Regeln
- Ports definieren Schnittstellen zwischen Schichten
- Adapters verbinden externe Systeme mit inneren Schichten
- Abhängigkeiten zeigen immer nach innen



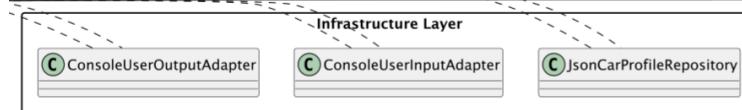
### Presentation Layer



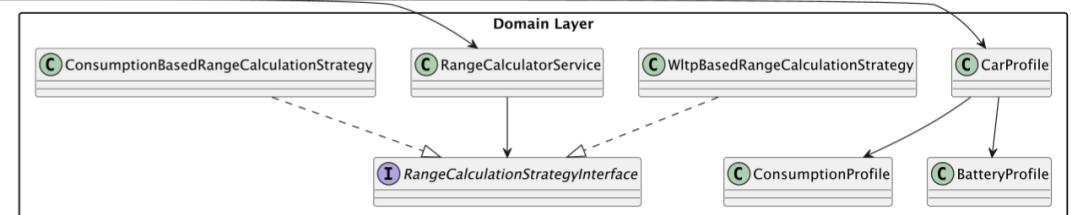
### Application Layer



### Infrastructure Layer



### Domain Layer



# **CLEAN ARCHITECTURE: ANALYSE U. BEGRÜNDUNG**

# DOMAIN LAYER

*Der Domain Layer ist das Herzstück der Clean Architecture und enthält die gesamte Geschäftslogik ohne externe Abhängigkeiten*

- **CarProfile** (Entity): Kernentität, die ein Car mit seinen Eigenschaften repräsentiert
- **BatteryProfile** (Value Object): Immutable Objekt, das die Batterieeigenschaften eines Fahrzeugs beschreibt
- **ConsumptionProfile** (Value Object): Kapselt Verbrauchsdaten und Verbrauchsfaktoren des Fahrzeugs ab
- **RangeCalculatorService** (Service): Berechnet die Reichweite basierend auf verschiedenen Strategien
- **RangeCalculationStrategyInterface**: Definiert die Schnittstelle für verschiedene Berechnungsalgorithmen

# APPLICATION LAYER

*Der Application Layer implementiert Use Cases und definiert Ports. Die Klassen orchestrieren Domänenobjekte, ohne direkten Zugriff auf die Infrastruktur, was dadurch die Dependency Rule einhält*

- **CalculateRangeUseCaseInterface** (Port): Definiert die Schnittstelle für den Anwendungsfall der Reichweitenberechnung
- **CalculateRangeInteractor** (Use Case): Implementiert die Geschäftslogik für die Reichweitenberechnung
- **CarProfilePersistencePortInterface** (Port): Abstrahiert die Datenpersistenz für Fahrzeugprofile
- **NavigationUseCase** (Use Case): Steuert die Navigationslogik und Routenplanung

# INFRASTRUCTURE LAYER

*Der Infrastructure Layer implementiert konkrete Adapter für externe Systeme. Diese Klassen sind austauschbar ohne Änderungen an inneren Schichten, was wichtig für e.g. OCP wichtig ist.*

- **JsonCarProfileRepository:** Implementiert die Persistenz von Fahrzeugprofilen als JSON-Dateien
- **ConsoleUserInputAdapter:** Verarbeitet Benutzereingaben über die Konsole
- **ConsoleUserOutputAdapter:** Zeigt Ausgaben und Ergebnisse in der Konsole an
- **ValidationUtils:** Hilfsmethoden zur Validierung von Eingabedaten

# PRESENTATION LAYER

*Der Presentation Layer kapselt die Benutzerinteraktion. Diese Klassen kommunizieren nur über definierte Schnittstellen mit inneren Schichten, was eine klare Trennung von UI und Geschäftslogik gewährleistet*

- **ApplicationControllerWithActionMenu:** Hauptcontroller, der die verschiedenen Aktionen der Anwendung steuert
- **RangeCalculationController:** Spezialisierter Controller für die Reichweitenberechnung
- **ActionMenuView:** Stellt das Hauptmenü der Anwendung dar
- **CarProfileView:** Visualisiert die Fahrzeugprofile und deren Details

# DOMAIN CODE (1P)

*Kurze Erläuterung in eigenen Worten, was Domain Code ist – 1 Beispiel im Code zeigen, das bisher noch nicht gezeigt wurde.*

- Bildet die Fachlogik der Domäne (Geschäftsregeln) ab
- Modelliert Prozesse, Regeln und Datenstrukturen
- Unabhängig von externen Systemen (Datenbank, UI, Frameworks,...)
- Bleibt Stabil bei Änderung (an Technologien)
- Domäne orientiert (Entity, Valueobject, Aggregate)
- Bleibt fachlich und ist frei von Presentation- und Persistenzlogik (siehe Domainlayer)

# BEISPIEL DOMAIN CODE: RANGECALCULATORSERVICE

```
public class RangeCalculatorService { // Reichweitenberechnung innerhalb der Domäne
    private final List strategies; // Hält verschiedene fachliche Berechnungsstrategien
    private RangeCalculationStrategyInterface defaultStrategy; // Definiert, welche Strategie die Standardberechnung ist

    public RangeCalculatorService() { // wird direkt mit zwei vordefinierten Strategien geladen
        this.strategies = new ArrayList<>();
        WltpBasedRangeCalculationStrategy wltpStrategy = new WltpBasedRangeCalculationStrategy();
        ConsumptionBasedRangeCalculationStrategy consumptionStrategy = new ConsumptionBasedRangeCalculationStrategy();
        this.strategies.add(wltpStrategy); // Beide Strategien werden in die Domain-intern verwendbare Liste übernommen
        this.strategies.add(consumptionStrategy);
        this.defaultStrategy = consumptionStrategy; // Standardmäßig wird die Verbrauchsstrategie benutzt
    }
}
```

# ANALYSE DES DOMAIN CODES

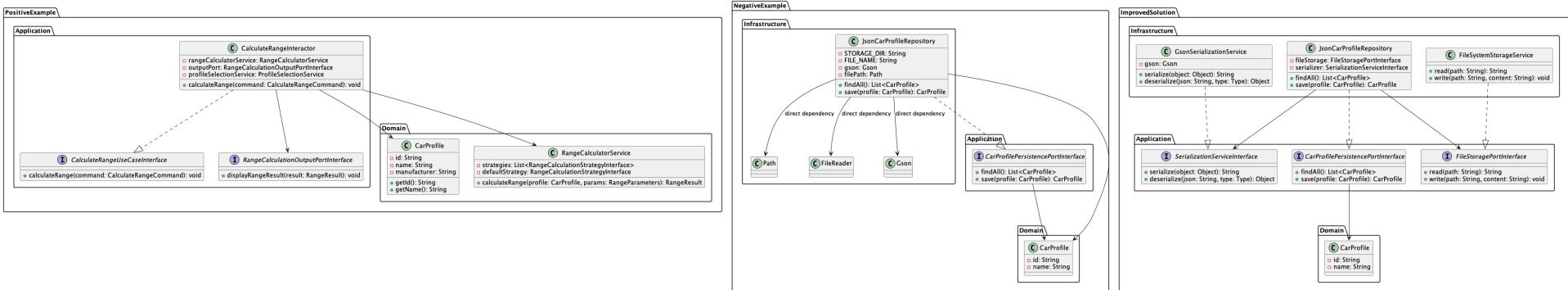
- Service kapselt reine Fachlogik zur Reichweitenberechnung
- Strategien sind fachliche Varianten zur Reichweitenermittlung
- Keine technische Infrastruktur wie DB, API oder UI enthalten
- Arbeitet nur mit Domain-spezifischen Typen (CarProfile, RangeParameters)
- Erweiterbarkeit durch neue Strategien innerhalb der Domäne möglich
- Berechnungen (calculateRange) folgen direkt fachlichen Regeln

# **ANALYSE DER DEPENDENCY RULE (3P)**

*In der Vorlesung wurde im Rahmen der 'Clean Architecture' die s.g. Dependency Rule vorgestellt. Je 1 Klasse zeigen, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule*

# ANALYSE DER DEPENDENCY RULE

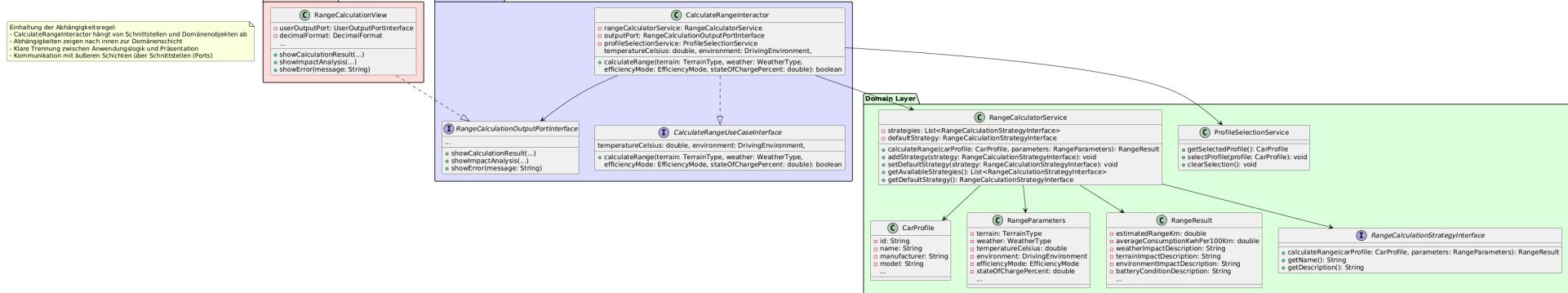
- Die Abhängigkeiten zeigen nur zum Zentrum der Architektur (größtenteils)
- Innere Schichten (Domain) dürfen nichts über äußere Schichten wissen
- Äußere Schichten (Infrastruktur) dürfen von inneren abhängen
- Die Abstraktionen (Interfaces) gehören zur inneren Schicht
- Die Implementierungen gehören zur äußeren Schicht



# POSITIVES BEISPIEL: CALCULATORANGEINTERACTOR

```
public class CalculateRangeInteractor implements CalculateRangeUseCaseInterface {  
  
    private final RangeCalculatorService rangeCalculatorService;  
    private final RangeCalculationOutputPortInterface outputPort;  
    private final ProfileSelectionService profileSelectionService;  
  
    public CalculateRangeInteractor(  
        RangeCalculatorService rangeCalculatorService,  
        RangeCalculationOutputPortInterface outputPort,  
        ProfileSelectionService profileSelectionService) {  
        this.rangeCalculatorService = Objects.requireNonNull(rangeCalculatorService);  
        this.outputPort = Objects.requireNonNull(outputPort);  
        this.profileSelectionService = Objects.requireNonNull(profileSelectionService);  
    }  
}
```

# UML: POSITIVES BEISPIEL



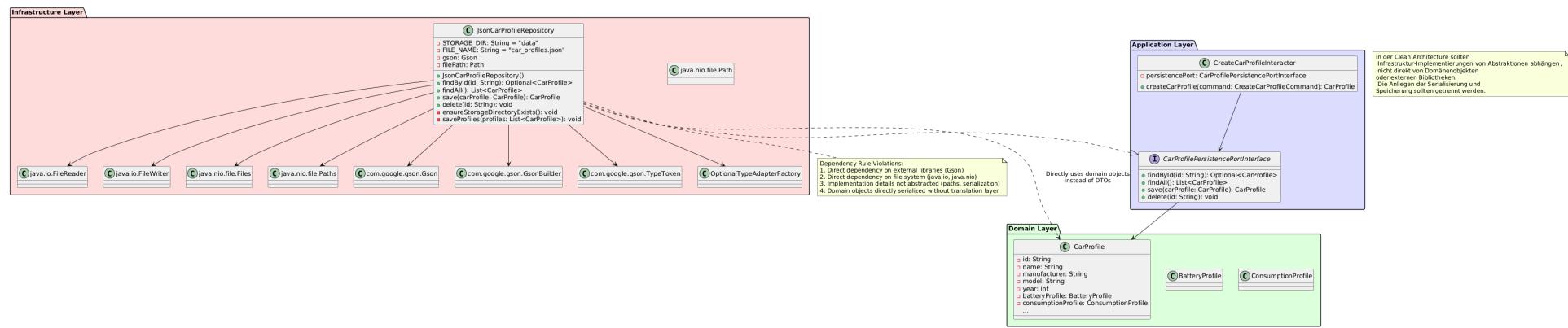
# ANALYSE DES POSITIVEN BEISPIELS

- Implementiert ein Interface aus der Application-Schicht
- Abhängigkeiten werden über Konstruktor-Injection bereitgestellt
- Abhängigkeiten sind alle Interfaces oder Domain-Services
- Keine Abhängigkeit zu konkreten Implementierungen der äußeren Schichten
- Ausgabe erfolgt über ein Port-Interface (RangeCalculationOutputPortInterface)
- Strikte Validierung mit Objects.requireNonNull()

# NEGATIVES BEISPIEL: JSONCARPROFILEREPOSITORY

```
public class JsonCarProfileRepository implements CarProfilePersistencePortInterface {  
    private static final String STORAGE_DIR = "data";  
    private static final String FILE_NAME = "car_profiles.json";  
    private final Gson gson;  
    private final Path filePath;  
  
    public JsonCarProfileRepository() {  
        this.gson = new GsonBuilder()  
            .setPrettyPrinting()  
            .registerTypeAdapterFactory(new OptionalTypeAdapterFactory())  
            .create();  
        this.filePath = Paths.get(STORAGE_DIR, FILE_NAME);  
        ensureStorageDirectoryExists();  
    }  
}
```

# UML: NEGATIVES BEISPIEL



# ANALYSE DES NEGATIVEN BEISPIELS

Verstöße gegen die Dependency Rule:

- Direkte Abhangigkeit zu externen Bibliotheken (Gson)
- Direkter Zugriff auf das Dateisystem (java.io, java.nio)
- Hartcodierte Pfade (STORAGE\_DIR, FILE\_NAME)
- Ausnahmebehandlung mit generischen Exceptions
- Direkte Implementierung der Dateioperationen

Verbesserungsvorschlag:

- Einführung eines FileStoragePortInterface
- Auslagerung der JSON-Serialisierung
- Konfigurierbare Pfade (z.B. uber Umgebungsvariablen)
- Fachliche Exceptions wie ProfileNotFoundException
- Strikte Trennung von Persistenz und Domanenmodell

# KAPITEL 3: SOLID (8P)

# SRP – SINGLE RESPONSIBILITY PRINCIPLE (3P)

- *Jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML).*
- *Def.:*  
*Jede Klasse sollte genau eine Verantwortlichkeit haben, d.h. einen einzigen Grund für Änderungen.*

# **POSITIVES BEISPIEL: RANGCALCULATORSERVICE**

## **VERANTWORTUNG DER KLASSE:**

- Verwaltung von Reichweiten-Berechnungsstrategien
- Bereitstellung einer einheitlichen Schnittstelle zur Reichweitenberechnung
- Delegation der Berechnung an die jeweils konfigurierte Strategie

## **WAS DIE KLASSE AUSDRÜCKLICH NICHT MACHT:**

- Interaktion mit dem Benutzer (Eingaben verarbeiten)
- Ausgabe oder Darstellung der Ergebnisse
- Laden oder Speichern von Fahrzeugdaten

Die Klasse erfüllt somit das SRP: Änderungen wären ausschließlich aufgrund interner Strategie-Logik oder Berechnung notwendig – nicht wegen externer Anliegen.

# POSITIVES BEISPIEL: CODE

```
public class RangeCalculatorService {  
    private final List strategies;  
    private RangeCalculationStrategyInterface defaultStrategy;  
  
    public RangeCalculatorService() {  
        this.strategies = new ArrayList<>();  
  
        WltpBasedRangeCalculationStrategy wltpStrategy = new WltpBasedRangeCalculationSt  
        ConsumptionBasedRangeCalculationStrategy consumptionStrategy = new ConsumptionBa  
  
        this.strategies.add(wltpStrategy);  
        this.strategies.add(consumptionStrategy);  
  
        this.defaultStrategy = consumptionStrategy;  
    }  
}
```

# **NEGATIVES BEISPIEL: APPLICATIONCONTROLLERWITHACTION MENU**

# NEGATIVES BEISPIEL: ERKLÄRUNG

## VERANTWORTLICHKEITEN:

- **Navigation:** Verwaltet Zustandsübergänge zwischen allen Bildschirmen
- **UI-Logik:** Verarbeitet Benutzereingaben aus mehreren Menüs
- **Profilmanagement:** Erstellen, Editieren und Löschen von Fahrzeugprofilen
- **View-Management:** Interaktion mit zahlreichen Views
- **Validierung:** Eingabeverifikation und Datensammlung

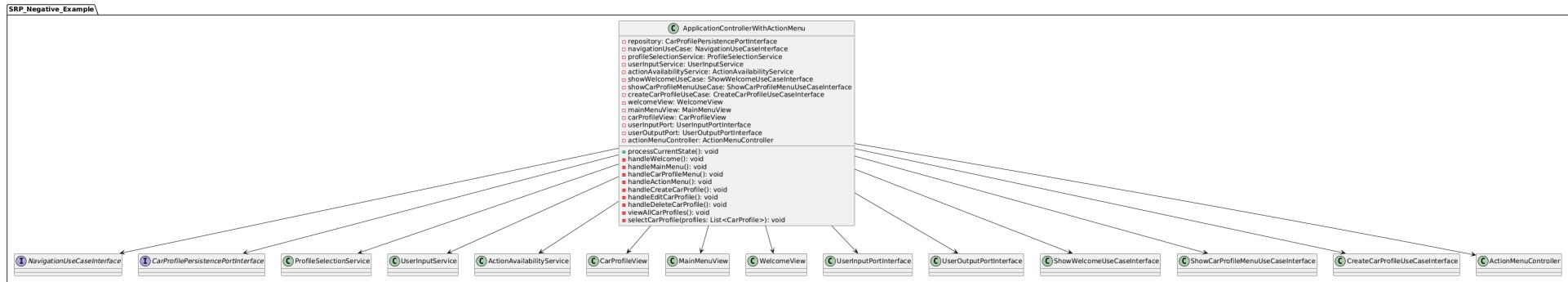
## PROBLEME:

- Mehrere Gründe für Änderungen (UI-Änderung, Navigationslogik, Profilmanagement)
- Schwer testbar (zu viele Abhängigkeiten und komplexe Interaktionen)
- Verletzt Clean Architecture durch zu enge Kopplung von Verantwortlichkeiten
- Hohe Komplexität durch zahlreiche Methoden und switch-case Logik
- Mehr als 300 Zeilen -> Schlechte Lesbarkeit

# NEGATIVES BEISPIEL: CODE

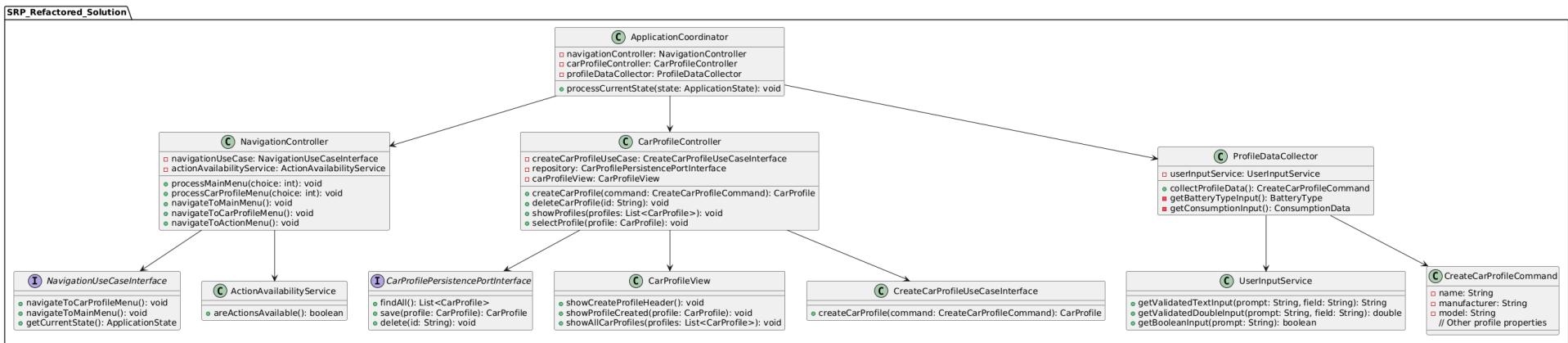
```
public class ApplicationControllerWithActionMenu {  
    private final CarProfilePersistencePortInterface repository;  
    private final NavigationUseCaseInterface navigationUseCase;  
    private final ProfileSelectionService profileSelectionService;  
    private final UserInputService userInputService;  
    private final ActionAvailabilityService actionAvailabilityService;  
  
    private final ShowWelcomeUseCaseInterface showWelcomeUseCase;  
    private final ShowCarProfileMenuUseCaseInterface showCarProfileMenuUseCase;  
    private final CreateCarProfileUseCaseInterface createCarProfileUseCase;  
  
    private final WelcomeView welcomeView;  
    private final MainMenuView mainMenuView;
```

# NEGATIVES BEISPIEL: UML



# NEGATIVES BEISPIEL: VERBESSERUNGEN & UML

- Klare Aufteilung in Verantwortlichkeitsbereiche (Controller, Collector, Views)
- Stärkere Modularität und Wiederverwendbarkeit
- Einfacheres Testen durch geringe Kopplung
- Einheitliche, kohärente Klassenstruktur  
-> Verletzt nicht die Clean Architecture mehr



# NEGATIVES BEISPIEL: REFACTORING

Aufteilung in spezialisierte Controller:

```
// Spezialisiert auf Navigation
public class NavigationController {
    private final NavigationUseCaseInterface
    private final ActionAvailabilityService a

    public void processMainMenu(int choice) {
        switch (choice) {
            case 1:
                navigationUseCase.navigateToCarPr
                break;
            case 2:
                if (actionService.areActionsAvail
                navigationUseCase.navigateToAction
            }
            break;
        }
    }
}
```

```
// Spezialisiert auf Profilmanagement
public class CarProfileController {
    private final CreateCarProfileUseCaseInte
    private final CarProfileView carProfileVi

    public CarProfile createCarProfile(Create
        CarProfile profile = createUseCase.cr
        carProfileView.showProfileCreated(pro
        return profile;
    }

    public void showProfiles(List profiles) {
        carProfileView.showAllCarProfiles(pro
    }
}
```

# NEGATIVES BEISPIEL: REFACTORING (FORTSETZUNG)

```
// Spezialisiert auf Eingabesammlung
public class ProfileDataCollector {
    private final UserInputService inputService;

    public CreateCarProfileCommand collectProfileData() {
        String name = inputService.getValidatedString("Enter name: ", "name");
        String manufacturer = inputService.getValidatedString("Enter manufacturer: ", "manufacturer");
        // Weitere Eingabeabfragen...

        return new CreateCarProfileCommand(
            name, manufacturer, model, year,
            hasHeatPump, wltpRange, dcPower,
            acPower, batteryType.name(), capacity);
    }
}
```

```
// Koordinator-Klasse mit weniger Verantwortung
public class ApplicationCoordinator {
    private final NavigationController navController;
    private final CarProfileController profileController;
    private final ProfileDataCollector dataCollector;

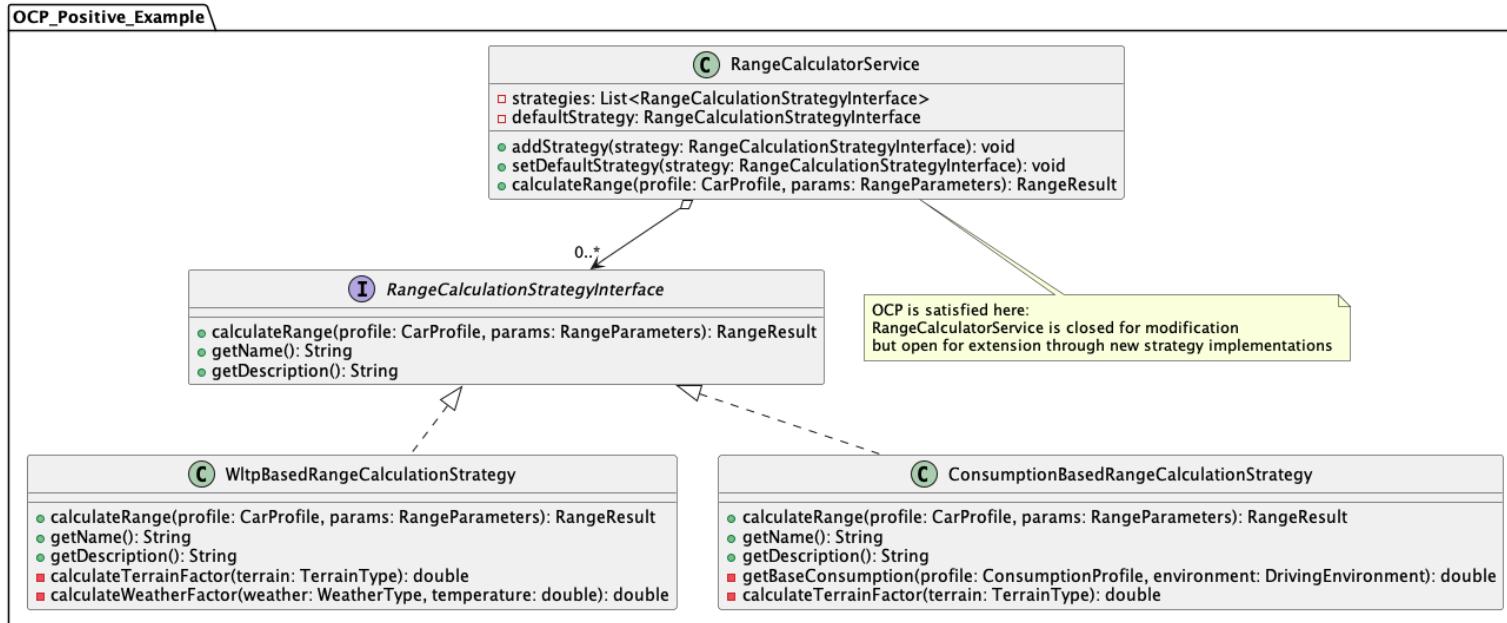
    public void processCurrentState(ApplicationState state) {
        switch (state) {
            case CREATE_CAR_PROFILE:
                CreateCarProfileCommand data =
                    dataCollector.collectProfileData();
                profileController.createCarProfile(data);
                navController.navigateToCarProfile();
                break;
            // Weitere vereinfachte Zustandsverarbeitung...
        }
    }
}
```

# OCP - OPEN CLOSED PRINCIPLE (3P)

*Jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?*

# **POSITIVES BEISPIEL: UML & BESCHREIBUNG**

- Abstraktion über Interface: RangeCalculationStrategyInterface
- Neue Strategien können ohne Änderung der Kernlogik hinzugefügt werden
- Der RangeCalculatorService bleibt unverändert
- Einfache Erweiterung um neue Berechnungsmethoden
- Geschlossen für Modifikation: Bestehender Code wird nicht geändert



# POSITIVES BEISPIEL: CODE

```
// Abstraktion (Interface)
public interface RangeCalculationStrategyInterface {
    RangeResult calculateRange(CarProfile carProfile, RangeParameters parameters);
    String getName();
    String getDescription();
}

// Erste Implementierung
public class WltpBasedRangeCalculationStrategy implements RangeCalculationStrategyInterface {
    @Override
    public RangeResult calculateRange(CarProfile carProfile, RangeParameters parameters)
```

# OCP - NEGATIVES BEISPIEL: ACCHARGINGCALCULATOR

# OCP – NEGATIVES BEISPIEL: ERKLÄRUNG

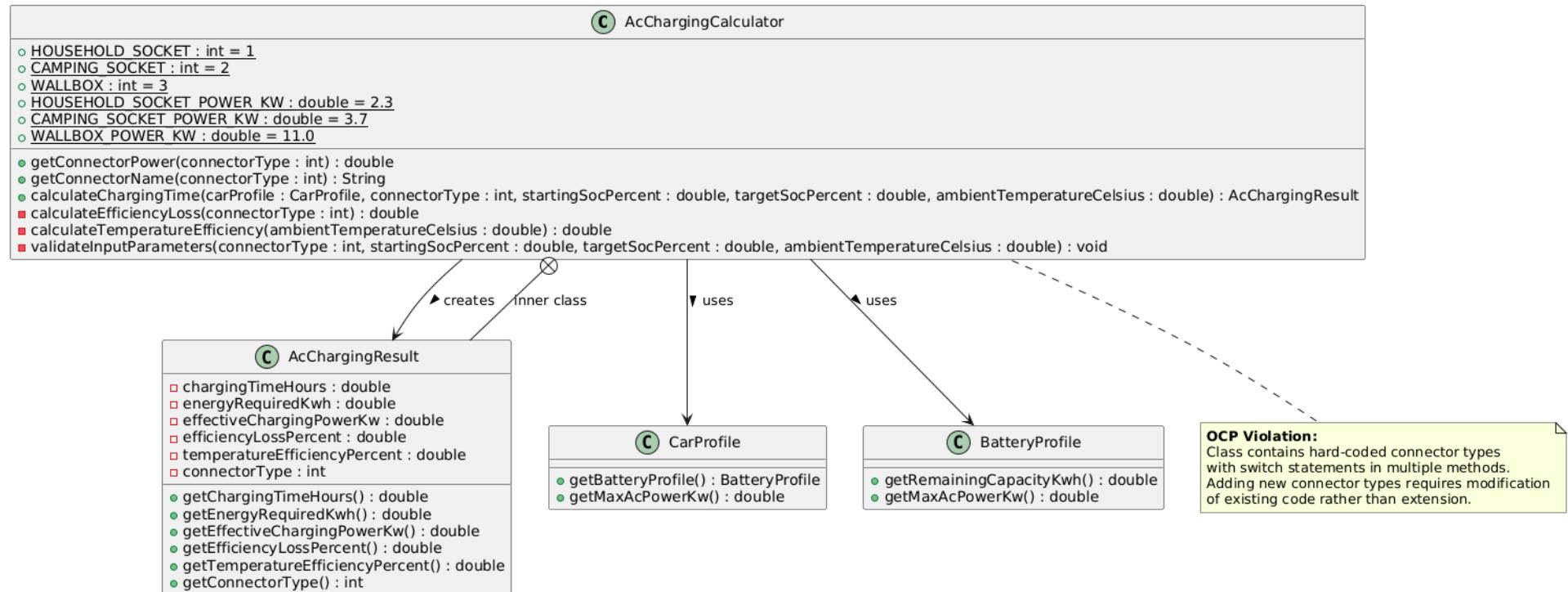
## VERANTWORTLICHKEITEN:

- **Typdefinition:**  
Verwaltet statische Konstanten für Steckertypen
- **Leistungsberechnung:**  
Bestimmt Ladeleistung abhängig vom Steckertyp
- **Bezeichnerlogik:**  
Liefert Namen abhängig vom Steckertyp
- **Effizienzberechnung:**  
Berechnet Effizienzverluste abhängig vom Steckertyp

## PROBLEME:

- Mehrere switch-Anweisungen für dieselbe Entscheidungslogik
- Verstoß gegen das Open/Closed Principle: Änderungen erfordern Modifikationen an bestehenden Methoden
- Neue Steckertypen erzwingen Änderungen an mehreren Stellen im Code
- Hohe Fehleranfälligkeit und schlechtere Testbarkeit
- Verletzung von DRY (Don't Repeat Yourself)

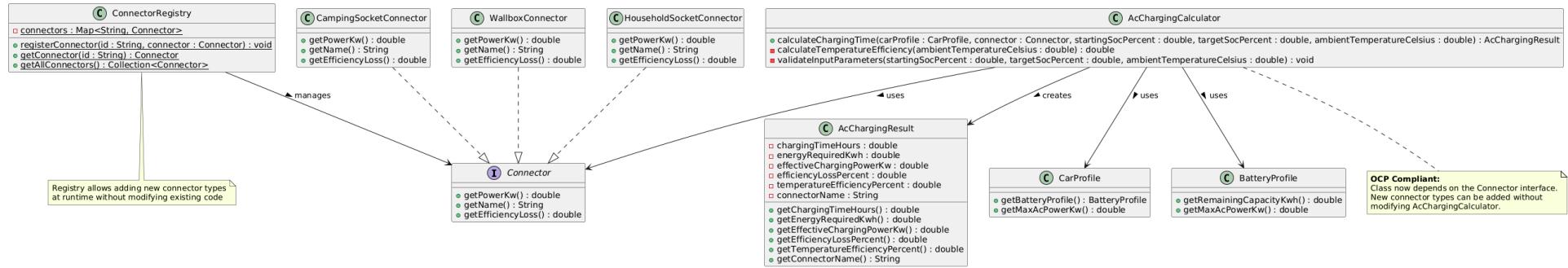
# NEGATIVES BEISPIEL: UML



# OCP – NEGATIVES BEISPIEL: CODE

```
public class AcChargingCalculator {  
    public static final double MIN_BATTERY_TEMPERATURE = -20.0;  
    public static final double MAX_BATTERY_TEMPERATURE = 60.0;  
    public static final double OPTIMAL_TEMPERATURE_MIN = 15.0;  
    public static final double OPTIMAL_TEMPERATURE_MAX = 35.0;  
  
    public static final int HOUSEHOLD_SOCKET = 1;  
    public static final int CAMPING_SOCKET = 2;  
    public static final int WALLBOX = 3;  
  
    public static final double HOUSEHOLD_SOCKET_POWER_KW = 2.3;  
    public static final double CAMPING_SOCKET_POWER_KW = 3.7;  
    public static final double WALLBOX_POWER_KW = 11.0;
```

# NEGATIVES BEISPIEL: LÖSUNG & UML



Lösungsansatz:

- Einführung eines Connector-Interfaces und konkreter Implementierungen
- Registry/Factory zur Verwaltung von Connector-Typen
- Neue Connector-Typen können ohne Änderung der AcChargingCalculator-Klasse hinzugefügt werden

# LSP - LISKOV SUBSTITUTION PRINCIPLE (2P)

- *Jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?*
- Eine abgeleitete Klasse soll an jeder beliebigen Stelle ihre Basisklasse ersetzen können, ohne, dass es zu unerwünschten Nebeneffekten kommt.

# POSITIVES BEISPIEL:

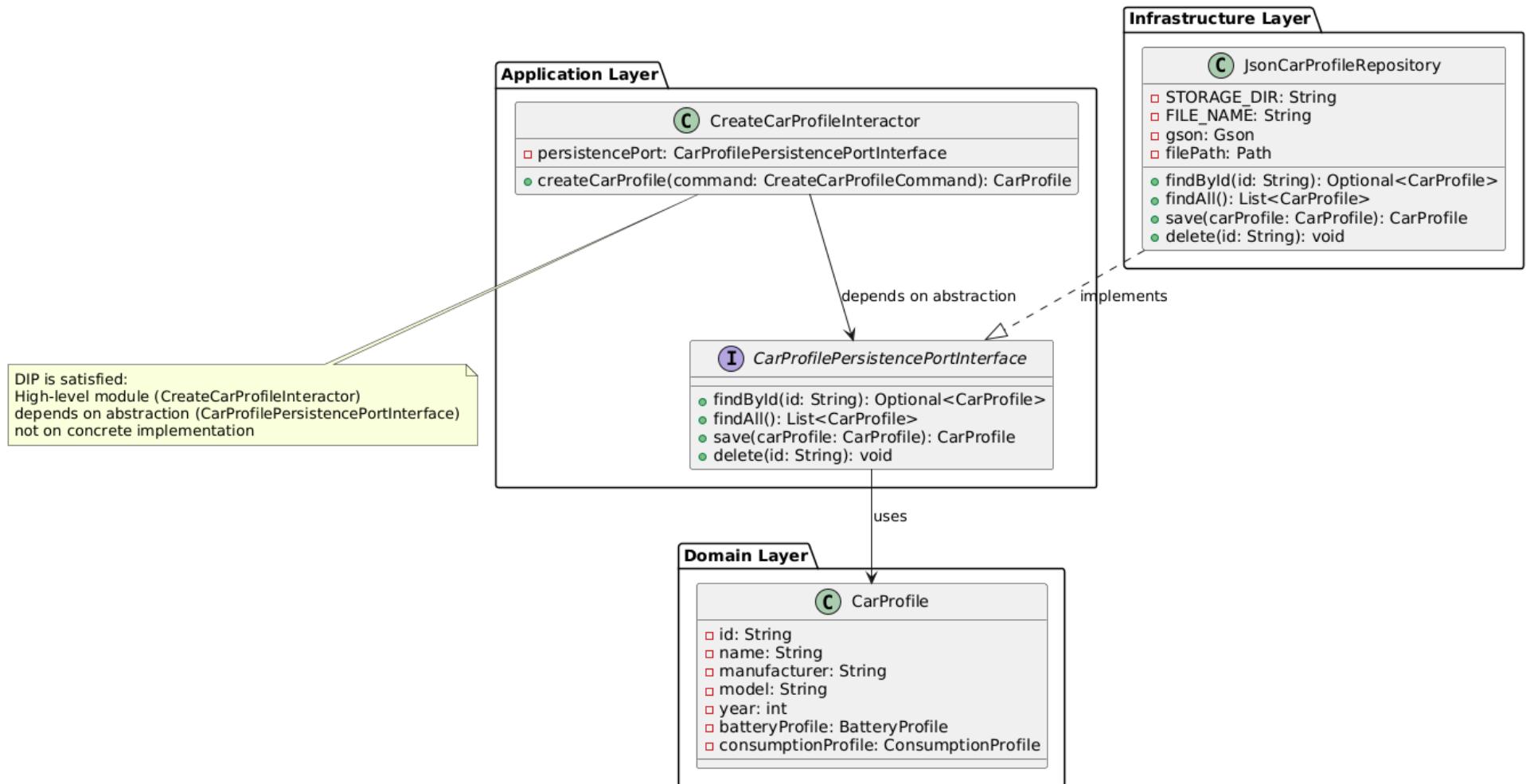
## HÄLT LSP EIN:

- Die Klasse `CreateCarProfileInteractor` verwendet ausschließlich das Interface `CarProfilePersistencePortInterface`
- `JsonCarProfileRepository` implementiert alle Methoden des Interfaces vollständig
- Das Verhalten bei Methodenaufrufen entspricht exakt dem was durch das Interface definiert wird
- Die konkrete Implementierung kann jederzeit durch eine andere ersetzt werden, ohne dass das Verhalten des Systems sich ändert.

## VORTEILE DADURCH:

- Ermöglicht sauberes und sicheres Austauschen z.B. der Persistenzlogik
- Erhöhte Wartbarkeit, da Änderungen in der Infrastruktur keine (Seiten-)Effekte auf die Anwendungsschicht haben
- Fördert Erweiterbarkeit und Offenheit für neue Anforderungen (Open/Closed-Prinzip).
- Entkopplung von Infrastruktur und Geschäftslogik führt zu robusterer Architektur.

# POSITIVES BEISPIEL: UML



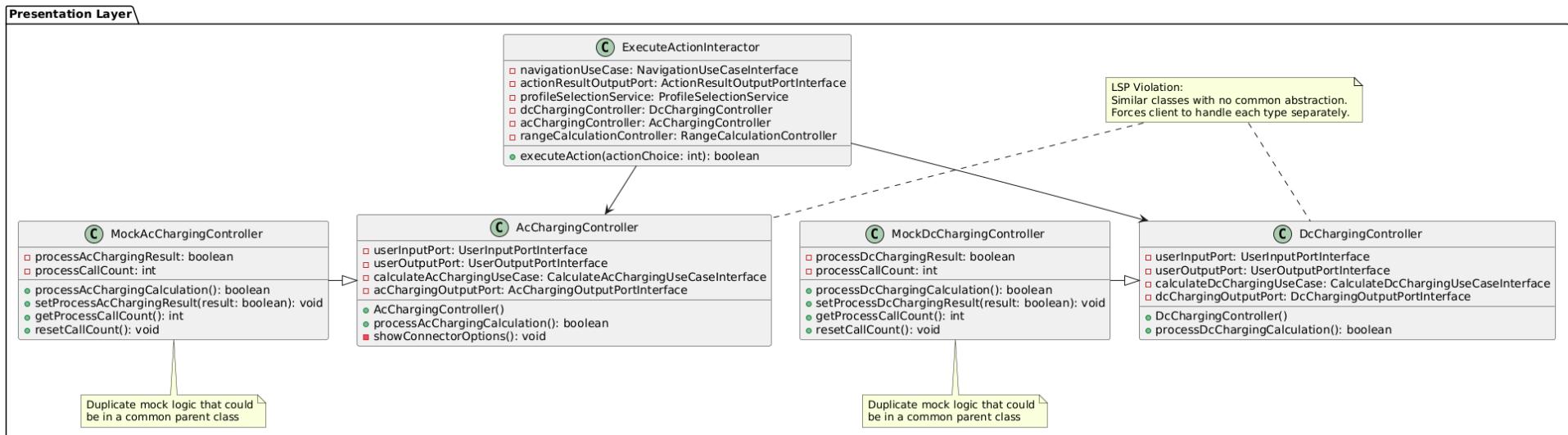
# **LSP – NEGATIVES BEISPIEL: CONTROLLER HIERARCHIE**

# LSP – NEGATIVES BEISPIEL: PROBLEME

- **Keine gemeinsame Basisklasse:** AC- und DC-Controller sind strukturell identisch, aber nicht gemeinsam abstrahiert.
- **Redundanz im Code:** Gleichartige Konstruktoren und Methoden wie `Process...Calculation()`, aber doppelt implementiert.
- **Verletzung der Austauschbarkeit:** Der `ExecuteActionInteractor` kann nicht einheitlich mit Controllern arbeiten.
- **Kein Polymorphismus:** Zwingt den Client zur Behandlung jeder Controller-Klasse separat (z. B. mit `if`-Verzweigungen).
- **Mocks dupliziert:** Separate Mock-Klassen mit nahezu identischem Code.

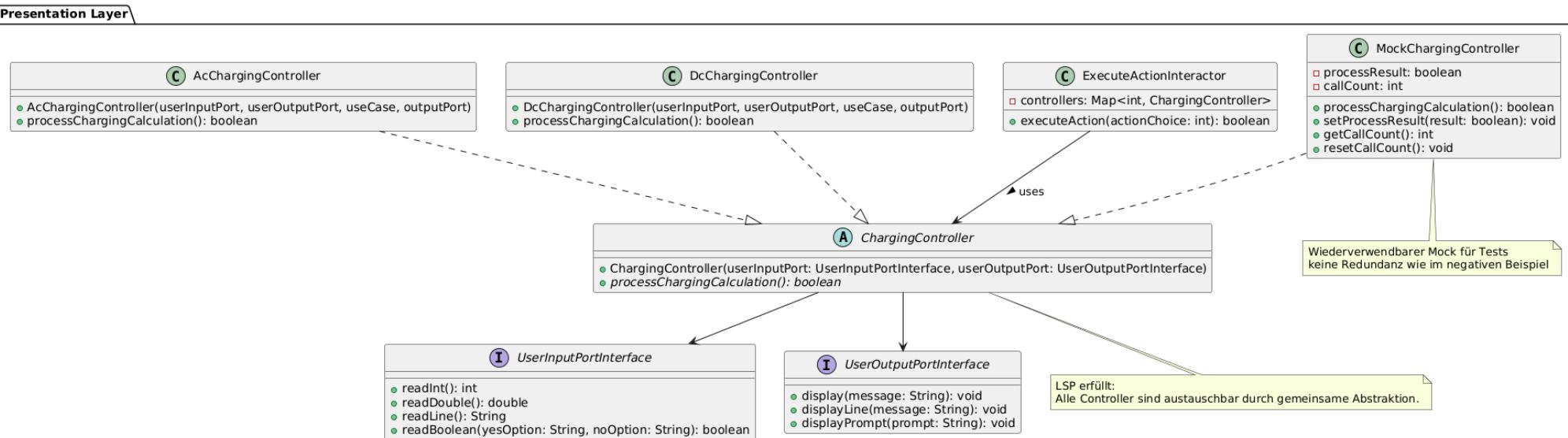
Diese Struktur verletzt das Liskov-Substitutionsprinzip, da die Controller nicht polymorph verwendet werden können, obwohl sie die gleiche grundlegende Funktionalität anbieten

# NEGATIVES BEISPIEL: UML



# NEGATIVES BEISPIEL: LÖSUNG & UML

- Einführung einer gemeinsamen Basisklasse `ChargingController` mit Methode `processChargingCalculation()`
- Polymorphismus durch Vererbung: AC- und DC-Controller erben von dieser Abstraktion und können austauschbar verwendet werden
- Zentrale Nutzung im Client: Der `ExecuteActionInteractor` referenziert nur die Abstraktion, nicht konkrete Typen
- MockController vereinheitlicht: Ein `MockChargingController` ersetzt mehrere spezialisierte Mocks.



# **KAPITEL 4: WEITERE PRINZIPIEN (8P)**

# **ANALYSE GRASP: GERINGE KOPPLUNG (3P)**

*Eine bis jetzt noch nicht behandelte Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt; es müssen auch die Aufrufer/Nutzer der Klasse berücksichtigt werden*

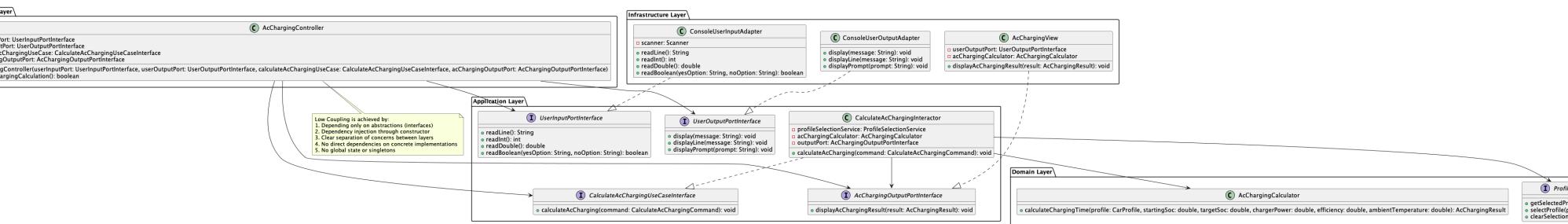
# **GERINGE KOPPLUNG: ACCHARGINGCONTROLLER**

Der AcChargingController koordiniert den Ablauf der AC-Ladeberechnung: Benutzereingaben werden gelesen, die Ladeparameter werden über ein Anwendungs & Use-Case Interface berechnet und die Ergebnisse werden anschließend ausgegeben

Die gesamte Kommunikation erfolgt ausschließlich über abstrahierte Schnittstellen (Ports), wodurch der Controller unabhängig von konkreten Implementierungen bleibt.

# GERINGE KOPPLUNG: UML & ERKLÄRUNG

- Verwendet nur Interfaces (Ports) statt konkreter Klassen
- Konstruktor-Injections aller Abhängigkeiten (DIP)
- Strikte Trennung von Steuerung, Ein-/Ausgabe und Fachlogik
- Keine hardcodierten Abhängigkeiten
- Alle Abhängigkeiten sind austauschbar und leicht testbar



# **GERINGE KOPPLUNG: AUFRUFER/NUTZER DER KLASSE**

Der `AcChargingController` wird durch eine externe Instanz, z. B. durch eine Main-Klasse oder ein Framework-Initializer, instanziert und diese übergibt die nötigen Ports per Konstruktor (Dependency Injection) und ruft die Methode `processChargingCalculation()` auf.

Somit bleibt der Controller vollständig entkoppelt vom Lebenszyklus und der konkreten Implementierung seiner Abhängigkeiten.

# GERINGE KOPPLUNG: VORTEILE DER GERINGEN KOPPLUNG

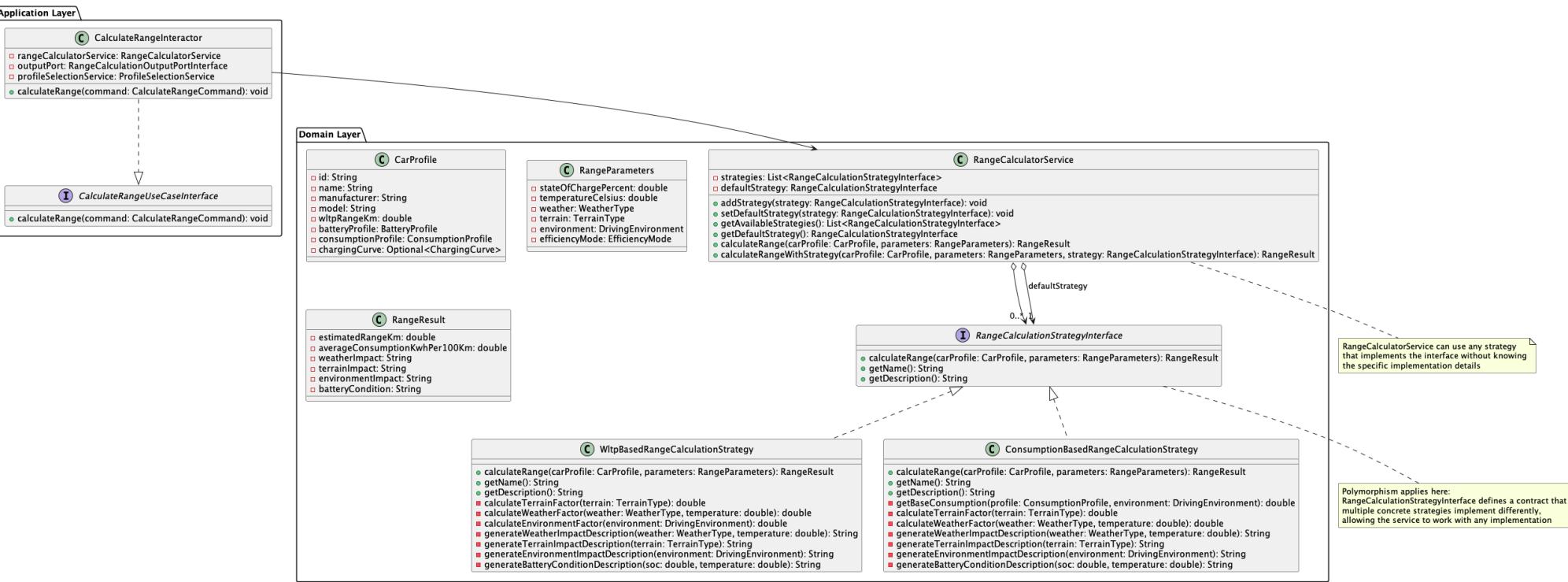
- **Testbarkeit:** Einfaches Mocking der Abhängigkeiten für Unit-Tests
- **Wartbarkeit:** Änderungen an Implementierungen der Ports haben keinen Einfluss auf den Controller
- **Flexibilität:** Verschiedene Implementierungen können ohne Änderung am Controller ausgetauscht werden
- **Wiederverwendbarkeit:** Der Controller kann mit unterschiedlichen Implementierungen genutzt werden
- **Verständlichkeit:** Klare Verantwortlichkeiten und Abhängigkeiten

# **ANALYSE GRASP: POLYMORPHISMUS (3P)**

*Eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt*

# RANGECALCULATIONSTRATEGY

- **Polymorphismus:** Verantwortung für das Verhalten wird an ein abstraktes Interface delegiert so dass konkrete Klassen polymorph interagieren können
- **Prinzip der Offenheit:** System ist offen für neue Strategien, aber geschlossen für Änderungen (OCP).
- **Geringe Kopplung:** Der aufrufende Code ist nicht abhängig von konkreten Implementierungen, sondern lediglich vom Interface.
- **Starke Kohäsion:** Jede Strategie kapselt exakt eine konkrete Berechnungslogik.



# VORTEILE DES POLYMORPHISMUS IM ANWENDUNGSKONTEXT

- **Erweiterbarkeit:** Neue Strategien können einfach hinzugefügt werden, ohne bestehende Klassen zu verändern.
- **Wiederverwendbarkeit:** Strategien sind unabhängig voneinander einsetzbar und modular.
- **Kapselung:** Die Implementierungsdetails der einzelnen Strategien sind vollständig vom aufrufenden Code abgeschirmt.
- **Wartbarkeit:** Änderungen an einer Strategie beeinflussen keine anderen Klassen, da jede Strategie isoliert ist.
- **Testbarkeit:** Einzelne Strategien können unabhängig getestet werden, was die Qualitätssicherung vereinfacht.
- **Komposition:** Die konkrete Strategie kann zur Laufzeit ausgetauscht oder kombiniert werden – ohne Anpassung des Client-Codes.

# **DRY – DON'T REPEAT YOURSELF (2P)**

*Ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen*

# DRY: CARPROFILECONTROLLER

Commit [89f9ea2](#):

Refactoring von [CarProfileController](#) zur Reduktion der duplizierten Logik

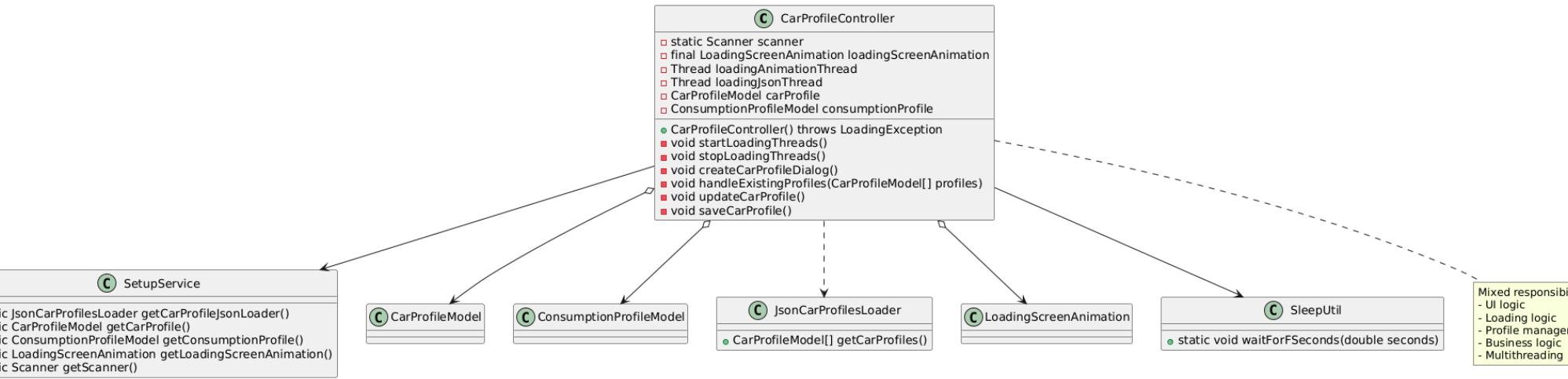
## PROBLEM:

- Duplizierter Code für ähnliche Operationen
- Schwer zu erkennendes Muster im Code
- Business-Logik war mit UI-Logik vermischt
- Hohe Fehleranfälligkeit bei Änderungen
- Clean Architecture war nicht erfüllt

## LÖSUNG:

- Auslagerung der Business-Logik in einen Service
- Zusammenfassung ähnlicher Methoden
- Einführung von Hilfsmethoden für gemeinsame Funktionalität
- Trennung von UI und Geschäftslogik
- Folgen von GRASP & SOLID

# DRY: ALTES UML

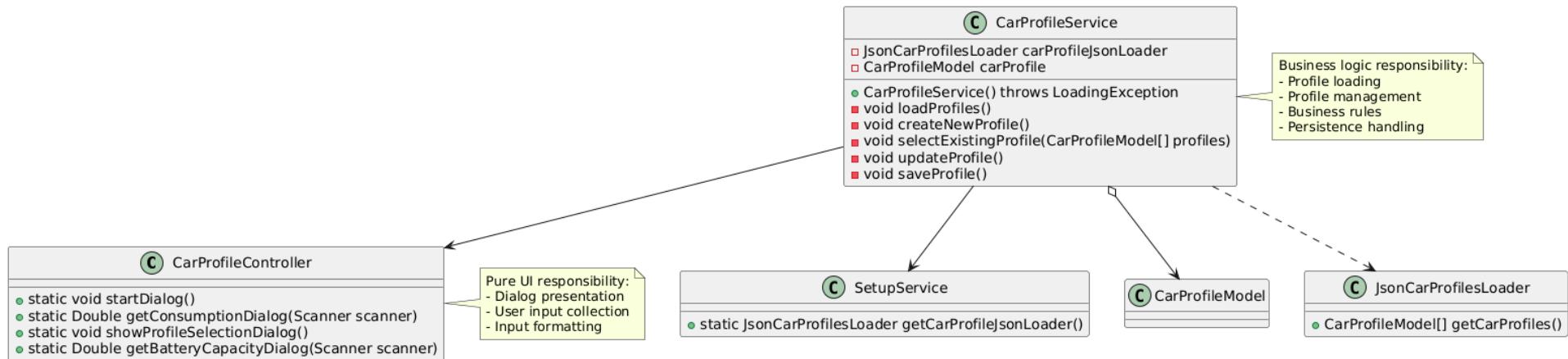


Mixed responsibi  
- UI logic  
- Loading logic  
- Profile manager  
- Business logic  
- Multithreading

# DRY – VORHER: CODE

```
public class CarProfileController {  
    private static Scanner scanner;  
    private final LoadingScreenAnimation loadingScreenAnimation;  
    private Thread loadingAnimationThread;  
    private Thread loadingJsonThread;  
    private CarProfileModel carProfile;  
    private ConsumptionProfileModel consumptionProfile;  
  
    public CarProfileController() throws LoadingException {  
        JsonCarProfilesLoader carProfileJsonLoader = SetupService.getCarProfileJsonLoa  
        carProfile = SetupService.getCarProfile();  
        consumptionProfile = SetupService.getConsumptionProfile();  
        loadingScreenAnimation = SetupService.getLoadingScreenAnimation();  
        loadingAnimationThread = new Thread(loadingScreenAnimation);  
        loadingJsonThread = new Thread(carProfileJsonLoader);  
    }  
}
```

# DRY: NEUES UML



# DRY – NACHHER: CODE

```
public class CarProfileController {  
    public static void startDialog() {  
        System.out.println(  
            "Hey! To begin with, let's see if you have any car profiles saved");  
    }  
  
    public static Double getConsumptionDialog(Scanner scanner) {  
        System.out.print("consumption: (in kWh) ");  
        String input = scanner.nextLine();  
        if (!input.isEmpty()) {  
            return Double.valueOf(input);  
        }  
        return null;  
    }  
}
```

# DRY: VORTEILE DES REFACTORINGS

- **Bessere Wartbarkeit:** Änderungen an UI oder Logik können unabhängig durchgeführt werden
- **Übersichtlichkeit:** Kürzere Klassen mit klaren Zuständigkeiten
- **Geringere Redundanz:** Gemeinsam genutzter Code ist an einer zentralen Stelle definiert
- **Bessere Testbarkeit:** Service kann unabhängig von UI getestet werden
- **Single Responsibility Principle:** Klassen haben nur eine Verantwortlichkeit
- **Clean Architecture:** Klare Trennung der Schichten zwischen Presentation und Application

# KAPITEL 5: UNIT TESTS (8P)

# **10 UNIT TESTS (2P)**

*Zeigen und Beschreiben von 10 Unit-Tests und Beschreibung, was getestet wird*

# UNIT TESTS

## TEST-STRATEGIE

- Fokus auf Domain-Logik als kritischen Kern
- Ignorieren von UI Elementen wie SOUTs
- ca. 100 automatisierte Tests (JUnit)
- 47% Gesamtdeckung (laut JaCoCo)
- Spezielle Tests für Randfälle und Fehlerbedingungen
- Mocks für externe Abhängigkeiten

## TESTARTEN IN ULRICA

- Unit-Tests für isolierte Klassen
- Integrationstests für Komponenteninteraktion
- Parametrisierte Tests für Datenvielfältigung
- Exception-Tests für Fehlerszenarien
- Stresstest für Berechnung mit großen Datenmengen

# TEST #1: DC CHARGING CALCULATOR

```
@Test  
    public void testCalculateChargingTime_WithChargingCurve() {  
        // Testwerte definieren  
        double startingSoc = 20.0;  
        double targetSoc = 80.0;  
        double maxStationPower = 250.0;  
        double ambientTemperature = 25.0;  
  
        // Ladekurve erstellen  
        Map curvePoints = new HashMap<>();  
        curvePoints.put(0.0, 150.0);    // 0% SOC -> 150 kW  
        curvePoints.put(20.0, 180.0);   // 20% SOC -> 180 kW  
        curvePoints.put(50.0, 100.0);   // 50% SOC -> 100 kW  
        curvePoints.put(80.0, 50.0);    // 80% SOC -> 50 kW  
        curvePoints.put(100.0, 10.0);   // 100% SOC -> 10 kW
```

Testet komplexe Ladeberechnungen mit dynamischer Ladekurve und verschiedenen SOC-Bereichen

# TEST #2: RANGE CALCULATOR SERVICE - STRATEGY PATTERN

```
@Test
public void testAddStrategy() {
    // Ausgangssituation dokumentieren
    int initialCount = rangeCalculatorService.getAvailableStrategies().size();

    // Mock-Strategie erstellen
    RangeCalculationStrategyInterface mockStrategy = new RangeCalculationStrategyI
        @Override
        public RangeResult calculateRange(CarProfile carProfile, RangeParameters par
            return new RangeResult(
                300.0,
                20.0,
                "No weather impact",
                "No terrain impact",
                "No environment impact",

```

Testet die dynamische Erweiterbarkeit durch das Strategy-Pattern im  
RangeCalculatorService

# TEST #3: CHARGINGCURVE INTERPOLATION

```
@Test  
  
    public void testGetChargingPowerAt_Interpolation() {  
        // Testdaten mit drei Kurven-Punkten erstellen  
        Map curvePoints = new HashMap<>();  
        curvePoints.put(0.0, 150.0);      // 0% -> 150 kW  
        curvePoints.put(50.0, 100.0);    // 50% -> 100 kW  
        curvePoints.put(100.0, 10.0);   // 100% -> 10 kW  
        ChargingCurve chargingCurve = new ChargingCurve(curvePoints);  
  
        // Test der linearen Interpolation zwischen Punkten  
  
        // Test Punkt zwischen 0% und 50%  
        assertEquals(125.0, chargingCurve.getChargingPowerAt(25.0), 0.001);  
  
        // Test Punkt zwischen 50% und 100%  
    }
```

Testet die korrekte lineare Interpolation bei der Ladekurve zwischen definierten Punkten

# TEST #4: EXECUTEACTION - KOMPLEXE UI-INTERAKTION

```
@Test  
public void testExecuteAction_NoProfileSelected() {  
    // Profil-Auswahl leeren  
    profileSelectionService.clearSelection();  
  
    // Action ausführen (DC-Ladung starten)  
    boolean result = executeActionInteractor.executeAction(1);  
  
    // Validieren des erwarteten Verhaltens  
    assertFalse(result);  
    assertEquals(1, expectedResultView.getErrorCount());  
    assertEquals(0, expectedResultView.getSuccessCount());  
    assertEquals(0, dcChargingController.getProcessCallCount());  
    assertEquals(0, acChargingController.getProcessCallCount());  
    assertEquals(0, rangeCalculationController.getProcessCallCount());
```

Testet das komplexe Zusammenspiel zwischen UI-Controller, Views und Geschäftslogik mit Mock-Objekten

# TEST #5: DC CHARGING - TEMPERATUREINFLUSS

```
@Test  
public void testCalculateChargingTime_LowTemperature() {  
    // Kalte Umgebungstemperatur als Testbedingung  
    double startingSoc = 20.0;  
    double targetSoc = 60.0;  
    double maxStationPower = 250.0;  
    double ambientTemperature = -10.0; // Sehr kalter Tag  
  
    // Service-Methode mit Testwerten aufrufen  
    DcChargingResult result = calculator.calculateChargingTime(  
        mockCarProfile,  
        startingSoc,  
        targetSoc,  
        maxStationPower,  
        ambientTemperature
```

Testet die realistische Simulation von Temperatureinflüssen auf das Ladeverhalten

# TEST #6: VALIDIERUNG DER EINGABEPARAMETER

```
@Test  
public void testValidateInputParameters_InvalidTargetSoc() {  
    // Ungültiger SOC-Wert (über 100%)  
    double startingSoc = 20.0;  
    double invalidTargetSoc = 110.0;  
  
    // Service-Methode aufrufen mit Erwartung auf Exception  
    assertThrows(IllegalArgumentException.class, () -> {  
        calculator.calculateChargingTime(  
            mockCarProfile,  
            startingSoc,  
            invalidTargetSoc,  
            250.0,  
            25.0  
    );  
}
```

Testet die robuste Validierung von Eingabeparametern in der Domain-Logik

# TEST #7: DI MIT MOCK-OBJEKTEN

```
@Test  
public void testExecuteAction_DcCharging() {  
    // Setup mit Mock-Objekten und Dependency Injection  
    profileSelectionService.selectProfile(testProfile);  
    dcChargingController.setProcessDcChargingResult(true);  
  
    // Aktion ausführen  
    boolean result = executeActionInteractor.executeAction(1);  
  
    // Validieren des Verhaltens und der Interaktionen  
    assertTrue(result);  
    assertEquals(0, actionResultView.getErrorCount());  
    assertEquals(1, dcChargingController.getProcessCallCount());  
    assertEquals(0, acChargingController.getProcessCallCount());  
    assertEquals(0, rangeCalculationController.getProcessCallCount());
```

Testet die korrekte Implementierung von Dependency Injection und das Zusammenspiel von Komponenten

# TEST #8: CHARGINGCURVE - ROBUSTHEIT

```
@Test  
  
    public void testInvalidConstructorArguments() {  
        // Test mit null-Wert  
        assertThrows(IllegalArgumentException.class, () -> {  
            new ChargingCurve(null);  
        });  
  
        // Test mit leerer Map  
        assertThrows(IllegalArgumentException.class, () -> {  
            new ChargingCurve(new HashMap<>());  
        });  
  
        // Test mit fehlenden Schlüsselpunkten  
        Map incompletePoints = new HashMap<>();  
        incompletePoints.put(50.0, 100.0);  
    }
```

Testet die Robustheit und Fehlerbehandlung bei ungültigen Eingabedaten

# TEST #9: PROFILESELECTIONSERVICE - STATUSVERWALTUNG

```
@Test  
public void testProfileSelection() {  
    // Initial kein Profil ausgewählt  
    assertFalse(profileSelectionService.hasSelectedProfile());  
  
    // Profil auswählen  
    CarProfile testProfile = new CarProfile.Builder()  
        .id("test-id")  
        .name("Test Car")  
        .build();  
    profileSelectionService.selectProfile(testProfile);  
  
    // Status und ausgewähltes Profil validieren  
    assertTrue(profileSelectionService.hasSelectedProfile());  
    assertEquals(testProfile, profileSelectionService.getSelectedProfile());
```

Testet die korrekte Statusverwaltung und Fehlerfälle im ProfileSelectionService

# TEST #10: VALIDIERUNG VON VALUE OBJECTS

```
@Test  
public void testValueObjectValidation() {  
    // Gültige Werte  
    BatteryProfile validProfile = new BatteryProfile(  
        BatteryType.LFP,  
        80.0,           // Kapazität in kWh  
        5.0,            // Degradation in Prozent  
        150.0,          // Max DC Ladeleistung  
        11.0            // Max AC Ladeleistung  
    );  
  
    // Eigenschaften validieren  
    assertEquals(BatteryType.LFP, validProfile.getType());  
    assertEquals(80.0, validProfile.getCapacityKwh(), 0.001);  
    assertEquals(5.0, validProfile.getDegradationPercent(), 0.001);
```

Testet die Validierung und Fehlerbehandlung bei der Erstellung von Value Objects

# **ATRIP: AUTOMATIC, THOROUGH UND PROFESSIONAL (2P)**

*je Begründung/Erläuterung, wie 'Automatic', 'Thorough' und 'Professional' realisiert wurde – bei 'Thorough' zusätzlich Analyse und Bewertung zur Testabdeckung]*

# ATRIP: AUTOMATIC

## AUTOMATISIERUNG DURCH CI/CD

- GitHub Actions Workflow für automatische Builds
- Automatische Testausführung bei jedem Push/PR
- Maven-basierte Build-Pipeline
- Automatische Code-Coverage-Analyse mit JaCoCo
- Automatische E-Mail-Benachrichtigungen über Testergebnisse

## AUTOMATISCHE TEST-TOOLS

- JUnit 4 für Unit Tests
- Maven Surefire für Testausführung
- CLOC für automatische Code-Analyse
- Automatische Validierung von Eingabeparametern
- Selbstvalidierung von Value Objects

# ATRIP: THOROUGH

## TESTABDECKUNG (COVERAGE)

- Gesamtdeckung: 47%
- Domain Layer: 68%
- Application Layer: 53%
- Presentation Layer: 42%
- Infrastructure Layer: 36%
- Core-Komponenten: 75%

## TESTARTEN

- Unit Tests für isolierte Komponenten
- Integrationstests für Zusammenspiel
- Parametrisierte Tests für Randfälle
- Exception Tests für Fehlerszenarien
- Mock-Tests für externe Abhängigkeiten

# ATRIP: PROFESSIONAL

## TESTORGANISATION

- Klare Teststruktur nach Clean Architecture
- Aussagekräftige Testbenennungen
- Separation of Concerns in Tests
- Wiederverwendbare Test-Utilities
- Dokumentierte Testfälle

## BEST PRACTICES

- Arrange-Act-Assert Pattern
- Given-When-Then Struktur
- Isolierte Testumgebungen
- Deterministische Tests
- Keine Test-Abhängigkeiten

# KLEINES ADD-ON ZU ATRP: MAILS

12:41 5G

mausio/ULRICA: Test Results and Code Coverage

Build Results

Build job nr 91 of **mausio/ULRICA** completed;  
Status: success

The job build has been triggered by mausio from a push

Test Summary

All tests are run automatically with each commit and daily schedule.

Test results and coverage reports are attached and available in GitHub Actions artifacts.

Code Quality

Line counting is performed automatically.

The CLOC report is attached.

**cloc\_output.txt**  
637 bytes

**index.html**  
12 KB

trash, folder, back, forward, edit

cloc\_output.txt

github.com/AlDanial/cloc v 1.98 T=0.04 s (1800.7 files/s, 104535.3 lines/s)

Language	files	blank	comment	code
Java	75	762	8	3584
SUM:	75	762	8	3584

index.html

ULRICA

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
<a href="#">org.ulrica.presentation.view</a>	0%	0%	75	75	288	288	59	59
<a href="#">org.ulrica.presentation.controller</a>	5%	0%	73	76	256	273	23	26
<a href="#">org.ulrica.domain.service</a>	73%	50%	100	172	99	354	14	65
<a href="#">org.ulrica.application.service</a>	19%	0%	29	40	54	67	6	17
<a href="#">org.ulrica.domain.valueobject</a>	76%	55%	44	118	25	184	10	60
<a href="#">org.ulrica</a>	0%	0%	3	3	35	35	2	2
<a href="#">org.ulrica.application.usecase</a>	60%	66%	14	38	59	146	10	26
<a href="#">org.ulrica.infrastructure.adapter</a>	0%	0%	13	13	29	29	9	9
<a href="#">org.ulrica.infrastructure.persistence</a>	81%	72%	6	25	14	67	2	16
<a href="#">org.ulrica.domain.entity</a>	98%	56%	14	47	0	71	0	31
<a href="#">org.ulrica.infrastructure.util</a>	100%	98%	1	42	0	33	0	12
<a href="#">org.ulrica.application.port.in</a>	100%	n/a	0	15	0	30	0	15
<a href="#">org.ulrica.domain</a>	100%	n/a	0	1	0	9	0	1
Total	3,535 of 6,782	47%	333 of 586	43%	372	665	859	1,586 135
								339
								21
								55

Created with [JaCoCo](#) 0.8.11.202310140853

# **FAKES UND MOCKS (4P)**

*Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fakes/Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks*

# **MOCK #1: MOCKPROFILESELECTIONSERVICE**

## **ZWECK, FUNKTIONALITÄT & IMPLEMENTIERUNG**

- Wird in Tests für Profilauswahl verwendet
- Simuliert die Auswahl eines Fahrzeugprofils ohne echte Persistenz
- Erlaubt gezielte Testszenarien für Abbruchbedingungen
- Trackt Aufrufe via Zählermethoden
- Implementiert das Interface `ProfileSelectionService`
- Erweitert um test-spezifische Hilfsmethoden

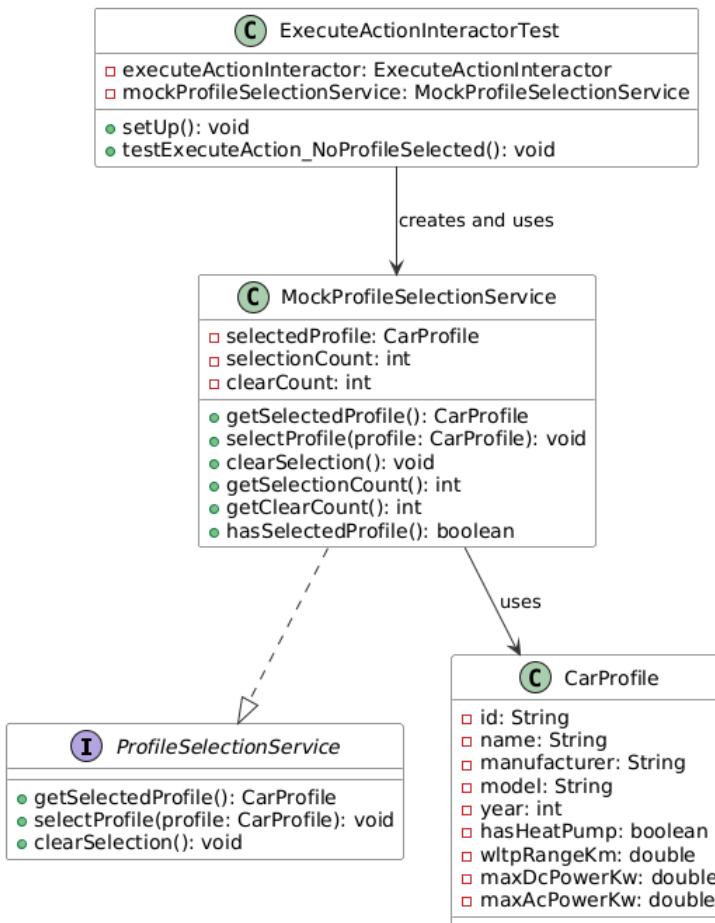
## **KRITIK VORAB:**

- Keine Validierung der Profilobjekte
- Kein Event-System für Änderungen → eingeschränkte Integrationstest-Fähigkeit

# MOCK #1: CODE

```
public class MockProfileSelectionService implements ProfileSelectionService {  
  
    private CarProfile selectedProfile;  
    private int selectionCount = 0;  
    private int clearCount = 0;  
  
    public MockProfileSelectionService() {  
        this.selectedProfile = null;  
    }  
  
    public MockProfileSelectionService(CarProfile initialProfile) {  
        this.selectedProfile = initialProfile;  
    }  
  
    @Override
```

# MOCK #1: UML-DIAGRAMM



# MOCK #2: MOCKUSERINPUTADAPTER

## FUNKTIONEN & IMPLEMENTIERUNG:

- Unterstützt verschiedene Eingabetypen
- Ermöglicht vordefinierte Eingabesequenzen
- Ermöglicht deterministische Testszenarien
- Ersetzt Benutzereingabe zur UI-Interaktion
- Bietet Fallback-Werte für leere Eingaben
- Implementiert `UserInputPortInterface`

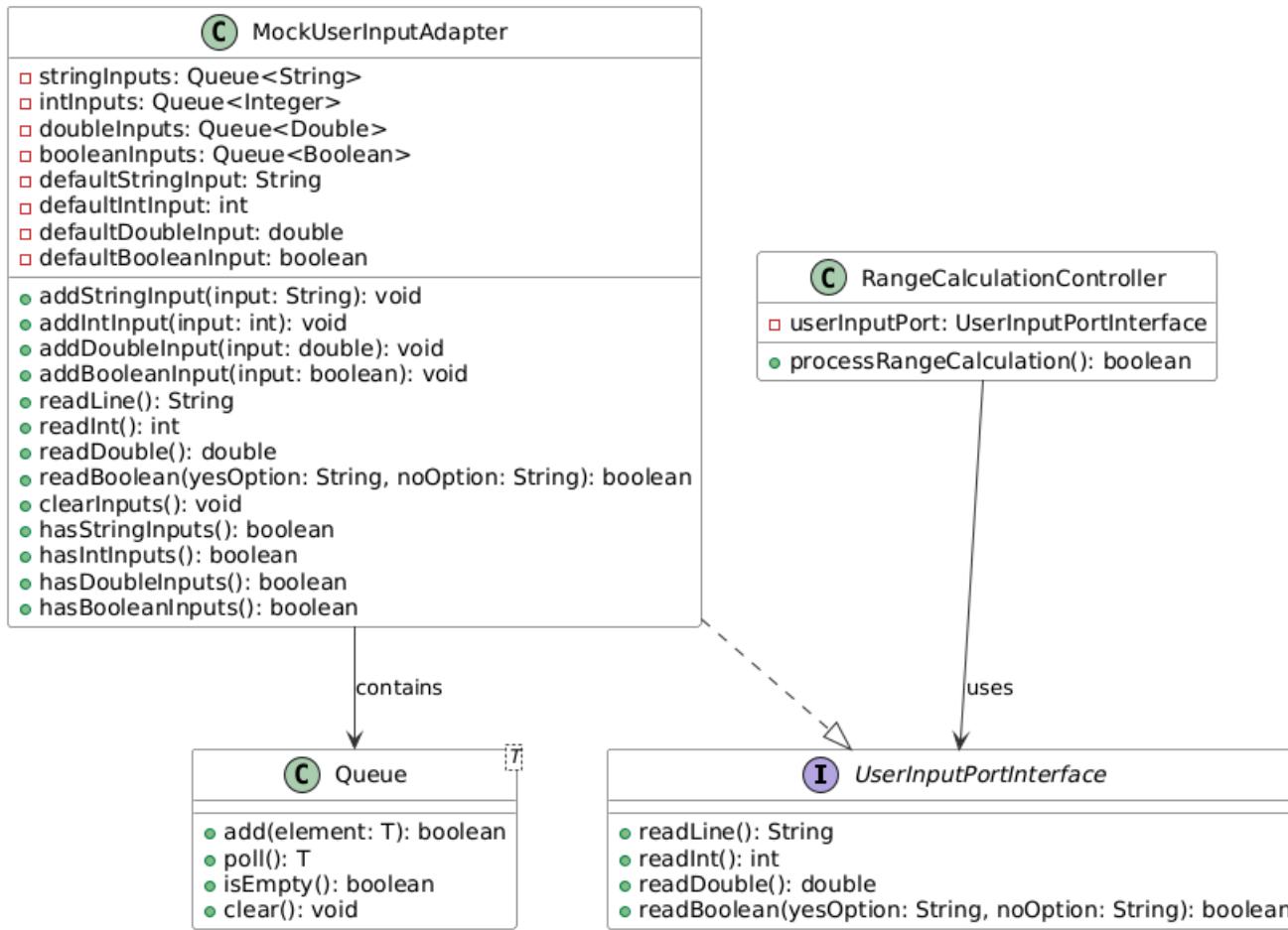
## KRITIK VORAB:

- Komplexität für einfache Tests relativ hoch
- Fehlendes Verhalten bei Fehleingaben (z.B. Exceptionhandling)

# MOCK #2: CODE

```
public class MockUserInputAdapter implements UserInputPortInterface {  
  
    private final Queue<String> stringInputs = new ArrayDeque<>();  
    private final Queue<Integer> intInputs = new ArrayDeque<>();  
    private final Queue<Double> doubleInputs = new ArrayDeque<>();  
    private final Queue<Boolean> booleanInputs = new ArrayDeque<>();  
  
    private String defaultStringInput = "";  
    private int defaultIntInput = 0;  
    private double defaultDoubleInput = 0.0;  
    private boolean defaultBooleanInput = false;  
  
    public void addStringInput(String input) {  
        stringInputs.add(input);  
    }
```

# MOCK #2: UML



# VERGLEICH DER MOCK-IMPLEMENTIERUNGEN

Aspekt	MockProfileSelectionService	MockUserInputAdapter
Zweck	Verhaltensverifikation (Methodenzähler)	Simulation von Eingabesequenzen
Komplexität	Niedrig (Status basiert)	Hoch (Typ-Queue-Handling)
Architektur-Schicht	Application / Domain	Infrastructure
Kritik	Keine Validierung, keine Events	Kein Fehlerverhalten, hoher Overhead

# **KAPITEL 6:**

# **DOMAIN DRIVEN DESIGN (DDD)**

## **(8P)**

# *INHALT DES KAPITELS*

## **1. UBIQUITOUS LANGUAGE (2P)**

- 4 Beispiele für die Ubiquitous Language
- Bezeichnung und Bedeutung
- Begründung der Zugehörigkeit

## **2. REPOSITORIES (1.5P)**

- Beschreibung des Repositories
- Code
- UML-Diagramm
- Begründung & Vorteile des Einsatzes

## **3. AGGREGATES (1.5P)**

- Beschreibung des Aggregates
- Code
- UML-Diagramm
- Begründung & Vorteile des Einsatzes

## **4. ENTITIES & VALUE OBJECTS (1.5P + 1.5P)**

- Beschreibung der Komponenten
- Code
- UML-Diagramme
- Begründung & Vorteile des Einsatzes

# **UBIQUITOUS LANGUAGE (2P)**

*Vier Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört*

# UBIQUITOUS LANGUAGE

Begriff	Bedeutung	Verwendung	Ubiquitous Language
SoC (State of Charge)	Prozentualer Ladezustand der Batterie (0-100%)	Parameter für Reichweitenberechnung und Ladezeit	Fachbegriff aus dem E-Mobility-Bereich, der die gemeinsame Sprache zwischen Technikern und Anwendern bildet
WLTP-Range	Standardisierte Reichweite nach <i>Worldwide Harmonized Light Vehicles Test Procedure</i>	Basis für realistische Reichweitenberechnung	Regulatorischer Begriff, der eine einheitliche Verständnisgrundlage für Fahrzeugleistung bietet

# UBIQUITOUS LANGUAGE (FORTSETZUNG)

Begriff	Bedeutung	Verwendung	Ubiquitous Language
Charging Curve	Verlauf der Ladeleistung in Abhängig vom SoC	Berechnung von Ladezeiten an DC-Ladestationen	Technisches Konzept, das sowohl für Entwickler als auch Benutzer die Ladeeigenschaften beschreibt
Battery Type (LFP, NMC, NCA)	Chemische Zusammensetzung der Batterie (Lithium-Eisenphosphat, Nickel-Mangan-Cobalt, Nickel-Cobalt-Aluminium)	Berechnung von Degradation und Ladegeschwindigkeit	Chemische Klassifikation, die technische Eigenschaften für das gesamte Team eindeutig kommunizierbar macht

# **REPOSITORIES (1.5P)**

*UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde*

# REPOSITORY: CARPROFILEREPOSITORY

- Persistenz von Fahrzeugprofilen ohne Offenlegung der Speichermechanismen
- Domain Entities können unabhängig von der Persistenz modelliert werden
- Unterstützt die "Illusion" einer *In-Memory-Sammlung*
- Ermöglicht einfachen Austausch der Persistenztechnologie im Falle von Refactoring

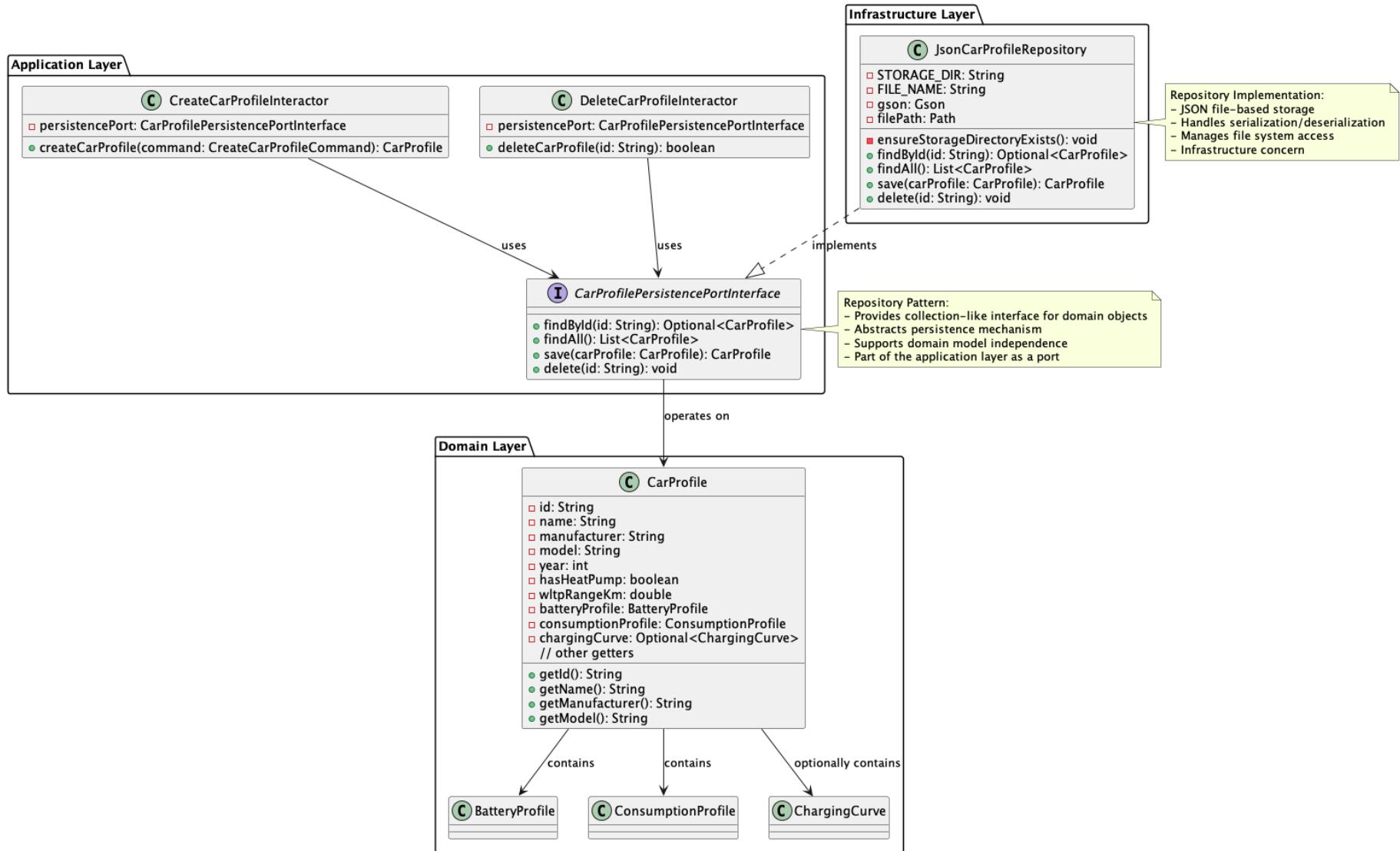
# REPOSITORY: CODE

```
//CarProfileInterface
public interface CarProfilePersistencePortInterface {
    Optional findById(String id);
    List findAll();
    CarProfile save(CarProfile carProfile);
    void delete(String id);
}

// Implementation
public class JsonCarProfileRepository implements CarProfilePersistencePortInterface {
    private static final String STORAGE_DIR = "data";
    private static final String FILE_NAME = "car_profiles.json";
    private final Gson gson;
    private final Path filePath;

    public JsonCarProfileRepository() {
        this(STORAGE_DIR, FILE_NAME);
    }
}
```

# REPOSITORY: UML



# **REPOSITORY: BEGRÜNDUNG & VORTEILE**

- Klare Trennung zwischen Domänenlogik und Persistenzmechanismus
- Vereinfachung des Domain Model durch Abstraktion der Persistenz
- Ermöglicht die Anwendung von Clean Architecture Prinzipien
- Erleichtert das Testen durch Mocking der Repository-Implementierung
- Entscheidung für Datenspeicherung in JSON-Dateien bleibt flexibel änderbar

# AGGREGATES (1.5P)

*UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde*

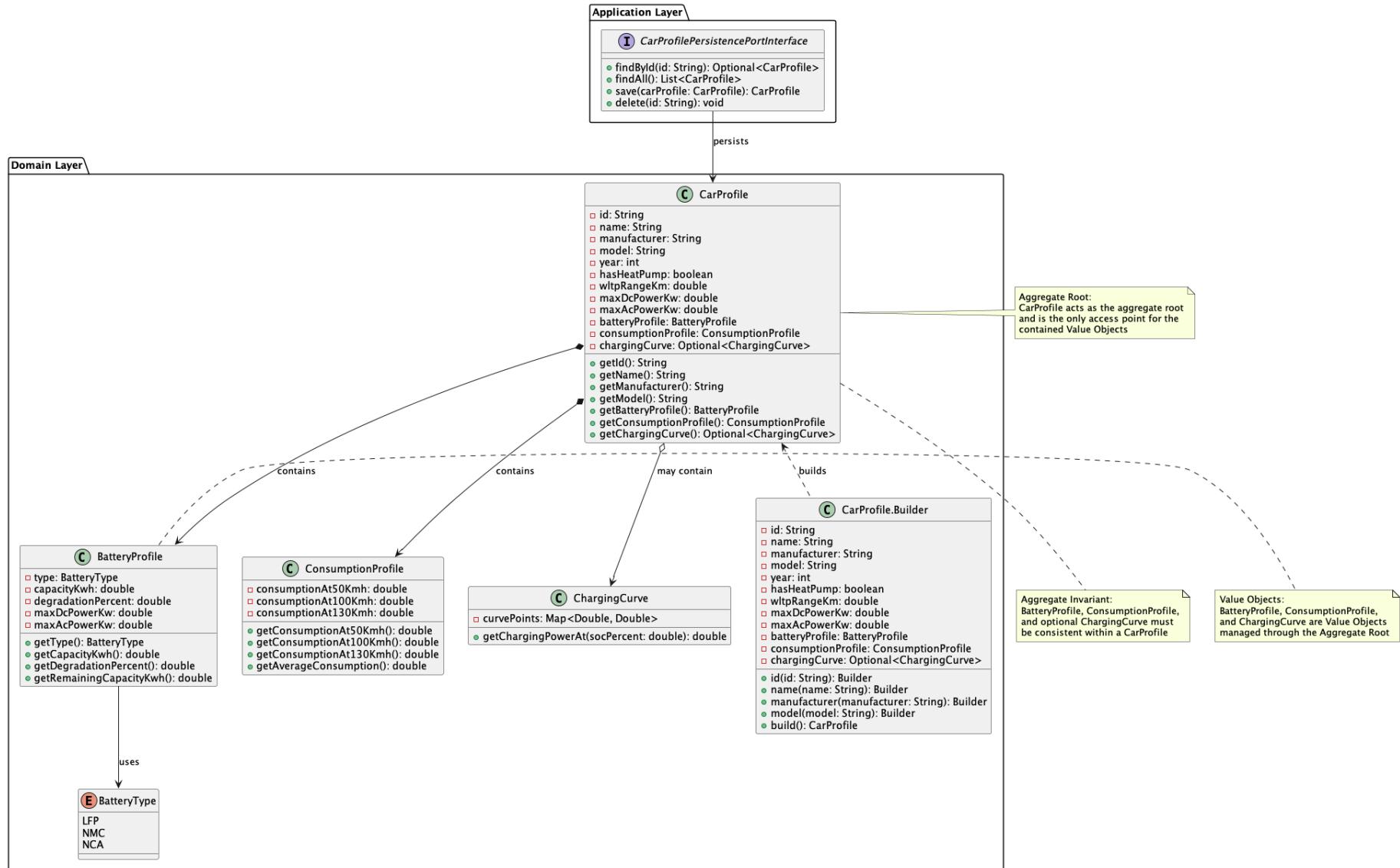
# AGGREGATES: CARPROFILE AGGREGATE ROOT

- Koordiniert mehrere Value Objects (BatteryProfile, ConsumptionProfile, ChargingCurve)
- Stellt die Konsistenz zwischen den Value Objects sicher
- Bildet eine sogenannte "transaktionale Einheit" für Änderungen
- Einziger Zugriffspunkt für enthaltene Value Objects
- Hat eine eigene Identität (ID) (-> Aggregate Root)

# AGGREGATES: CODE

```
public class CarProfile {  
    private final String id;  
    private final String name;  
    private final String manufacturer;  
    private final String model;  
    private final int year;  
    private final boolean hasHeatPump;  
    private final double wltpRangeKm;  
    private final double maxDcPowerKw;  
    private final double maxAcPowerKw;  
    private final BatteryProfile batteryProfile;  
    private final ConsumptionProfile consumptionProfile;  
    private final Optional chargingCurve;  
  
    private CarProfile(Builder builder) {  
        this.id = builder.id;  
        this.name = Objects.requireNonNull(builder.name, "Name cannot be null");  
        this.manufacturer = Objects.requireNonNull(builder.manufacturer, "Manufacturer cannot be null");  
        this.model = Objects.requireNonNull(builder.model, "Model cannot be null");  
        this.year = builder.year;  
        this.wltpRangeKm = builder.wltpRangeKm;  
        this.maxDcPowerKw = builder.maxDcPowerKw;  
        this.maxAcPowerKw = builder.maxAcPowerKw;  
        this.batteryProfile = builder.batteryProfile;  
        this.consumptionProfile = builder.consumptionProfile;  
        this.chargingCurve = builder.chargingCurve;  
        this.hasHeatPump = builder.hasHeatPump;  
    }  
  
    public static Builder builder() {  
        return new Builder();  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getManufacturer() {  
        return manufacturer;  
    }  
  
    public String getModel() {  
        return model;  
    }  
  
    public int getYear() {  
        return year;  
    }  
  
    public boolean hasHeatPump() {  
        return hasHeatPump;  
    }  
  
    public double getWltpRangeKm() {  
        return wltpRangeKm;  
    }  
  
    public double getMaxDcPowerKw() {  
        return maxDcPowerKw;  
    }  
  
    public double getMaxAcPowerKw() {  
        return maxAcPowerKw;  
    }  
  
    public BatteryProfile getBatteryProfile() {  
        return batteryProfile;  
    }  
  
    public ConsumptionProfile getConsumptionProfile() {  
        return consumptionProfile;  
    }  
  
    public Optional getChargingCurve() {  
        return chargingCurve;  
    }  
}
```

# AGGREGATE: UML





# AGGREGATES: VORTEILE & BEGRÜNDUNG

- CarProfile als Aggregate Root koordiniert die Konsistenz seiner Value Objects
- Stellt sicher, dass BatteryProfile, ConsumptionProfile und ChargingCurve immer konsistent sind
- Vereinfacht die Persistenz, da nur der Aggregate Root direkt gespeichert werden muss
- Schützt die Value Objects vor unerlaubten Änderungen von außen
- Modelliert die natürliche Gruppierung von Fahrzeugdaten und deren Komponenten

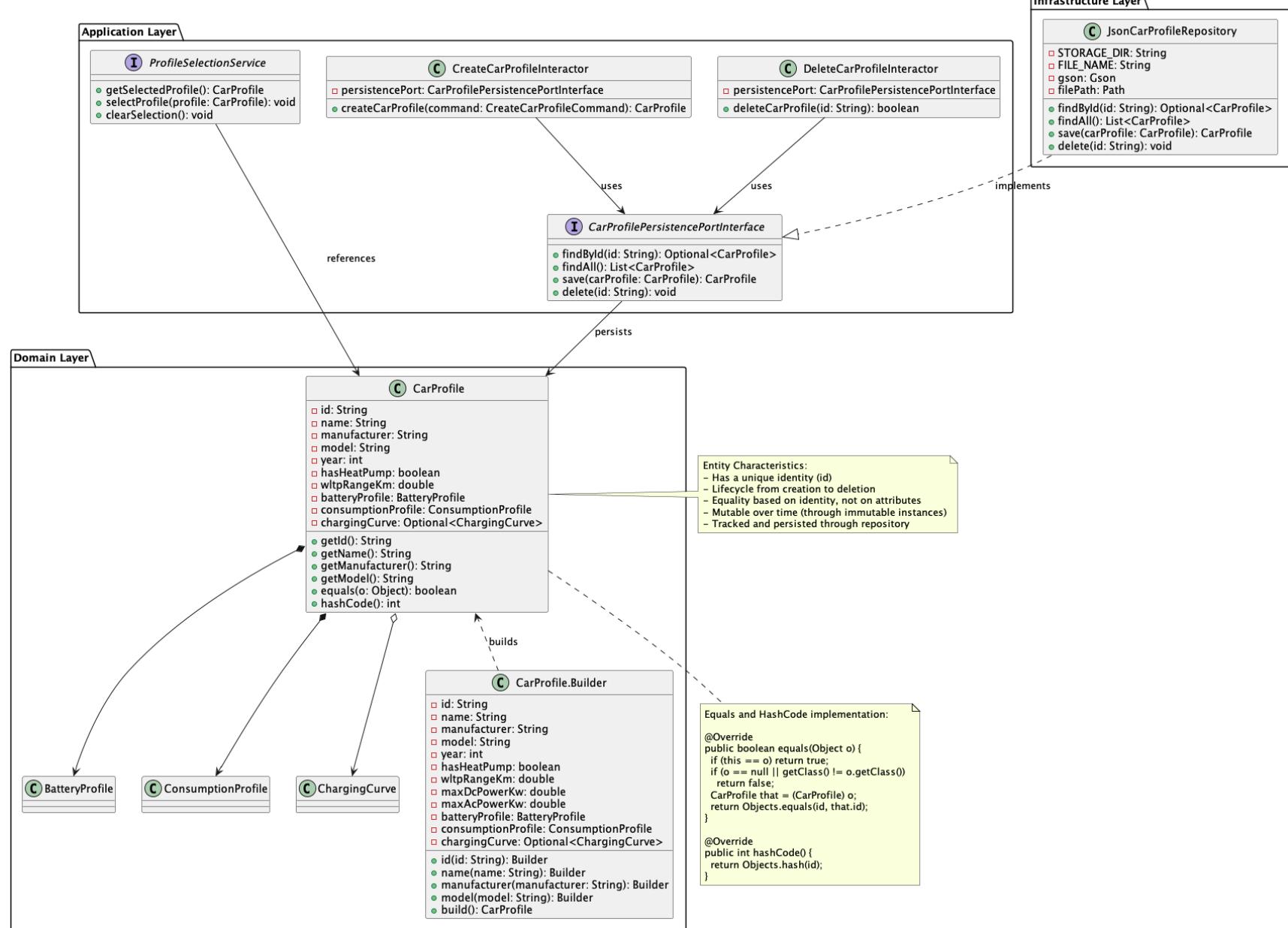
# **ENTITIES (1.5P)**

*UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keine geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde*

# ENTITIES: CARPROFILE ALS ENTITY

- Besitzt eine eindeutige Identität (UUID)
- Identität bleibt über den gesamten Lebenszyklus erhalten
- Kann verändert und weiterentwickelt werden (auch wenn in ULRICA immutable)
- Wird über die Identität verglichen, nicht über Attributwerte
- Repräsentiert ein reales Objekt (Elektrofahrzeug) mit eigenem Lebenszyklus

# ENTITY: UML



# ENTITY: BEGRÜNDUNG & VORTEILE

- Eindeutige Identifikation von Objekten unabhängig von ihren Attributen
- Möglichkeit, Entitäten über ihren Lebenszyklus hinweg zu verfolgen
- Konsistente Identifikation in verschiedenen Kontexten des Systems
- Klare Unterscheidung zwischen Identität und Attributen
- Natürliche Abbildung von realen Objekten mit eigener Identität

# VALUE OBJECTS (1.5P)

*UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist – NICHT, warum es nicht implementiert wurde*

# VALUE OBJECTS: BATTERYPROFILE ALS VALUEOBJECT

- Keine eigene Identität
- Immutable (unveränderlich) nach Erstellung
- Vergleich basiert auf allen Attributwerten
- Beschreibt eine Messung oder ein Konzept
- Kann problemlos ersetzt werden
- Validiert sich selbst bei der Erstellung

# VALUE OBJECT: CODE

```
public final class BatteryProfile {  
    private final BatteryType type;  
    private final double capacityKwh;  
    private final double degradationPercent;  
    private final double maxDcPowerKw;  
    private final double maxAcPowerKw;  
  
    public BatteryProfile(BatteryType type, double capacityKwh, double degradationPercent,  
        if (capacityKwh <= 0) {  
            throw new IllegalArgumentException("Battery capacity must be positive");  
        }  
        if (degradationPercent < 0 || degradationPercent > 100) {  
            throw new IllegalArgumentException("Degradation must be between 0 and 100 percent");  
        }  
        if (maxDcPowerKw <= 0) {  
            throw new IllegalArgumentException("Max DC power must be positive");  
        }  
        if (maxAcPowerKw <= 0) {  
            throw new IllegalArgumentException("Max AC power must be positive");  
        }  
    }  
}
```

# VALUE OBJECTS: UML



# VALUE OBJECTS: BEGRÜNDUNG & VORTEILE

- Batteryprofile und ConsumptionProfile sind immutable und damit thread-safe
- Vereinfachte Validierung durch Konstruktor basierte Erstellung
- Natürliche Modellierung von Messwerten und Konzepten ohne Identität
- Einfache Vergleichbarkeit durch attribut-basierte equals-Implementierung
- Können problemlos ausgetauscht werden, da sie keine Identität haben

# **KAPITEL 7: REFACTORING (8P)**

# *INHALT DES KAPITELS*

## **1. CODE SMELLS (2P)**

- Lange Methode in "RangeCalculationController"
- Duplizierter Code in "Charging Calculators"

## **2. REFACTORINGS (6P)**

- "Extract Method"
- Ersetzen von Conditionals mit Polymorphism

# **CODE SMELLS (2P)**

*Jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells (die benannt werden müssen) aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)*

# CODE SMELL 1: LANGE METHODE

```
/* public class RangeCalculationController {  
    private final CalculateRangeUseCaseInterface calculateRangeUseCase;  
    private final UserInputPortInterface userInputPort;  
    private final RangeCalculationOutputPortInterface outputPort;  
  
    public RangeCalculationController(  
        CalculateRangeUseCaseInterface calculateRangeUseCase,  
        UserInputPortInterface userInputPort,  
        RangeCalculationOutputPortInterface outputPort) {  
        this.calculateRangeUseCase = Objects.requireNonNull(calculateRangeUseCase, "Calculate range  
        this.userInputPort = Objects.requireNonNull(userInputPort, "User input port cannot be null")  
        this.outputPort = Objects.requireNonNull(outputPort, "Output port cannot be null");  
    } */  
  
    public boolean processRangeCalculation() {  
        try {  
            outputPort.showRangeCalculationHeader();  
        } catch (Exception e) {  
            logger.error("Error occurred during range calculation: " + e.getMessage());  
        }  
    }  
}
```

# CODE SMELL 1: ANALYSE

- Die Methode ist zu lang und komplex -> Unübersichtlich
- Viele verschiedene Verantwortlichkeiten in einer Methode
- Schlechte Lesbarkeit und Wartbarkeit
- Schwer zu testen
- Verletzt das Single Responsibility Principle

# CODE SMELL 1: LÖSUNGSVORSCHLAG

```
public boolean processRangeCalculation() {  
    try {  
        outputPort.showRangeCalculationHeader(); // Ausgabe eines Headers vor der Berechnung  
  
        RangeParameters parameters = collectRangeParameters(); // Sammeln aller Eingabewerte in ein Objekt  
        return calculateRangeUseCase.calculateRange(parameters); // Aufruf der Reichweitenberechnung mit den Werten  
  
    } catch (InvalidInputException e) {  
        outputPort.showError("Invalid input: " + e.getMessage()); // Ausgabe eines Fehlers bei ungültigen Eingaben  
        return false;  
    } catch (Exception e) {  
        outputPort.showError("Unexpected error: " + e.getMessage()); // Allgemeine Fehlerbehandlung  
        return false;  
    }  
}  
  
private RangeParameters collectRangeParameters() {  
    return new RangeParameters({
```

# CODE SMELL 2: DUPLICATE CODE

```
// DcChargingCalculator
private double calculateTemperatureEfficiencyFactor(double batteryTemperatureCelsius) {
    if (batteryTemperatureCelsius < MIN_BATTERY_TEMPERATURE ||
        batteryTemperatureCelsius > MAX_BATTERY_TEMPERATURE) {
        return 0.0;
    }

    if (batteryTemperatureCelsius >= OPTIMAL_TEMPERATURE_MIN &&
        batteryTemperatureCelsius <= OPTIMAL_TEMPERATURE_MAX) {
        return 1.0;
    }

    if (batteryTemperatureCelsius < OPTIMAL_TEMPERATURE_MIN) {
        return 0.5 + 0.5 * (batteryTemperatureCelsius - MIN_BATTERY_TEMPERATURE) /
            (OPTIMAL_TEMPERATURE_MIN - MIN_BATTERY_TEMPERATURE);
    } else {
        return 0.5 + 0.5 * (MAX_BATTERY_TEMPERATURE - batteryTemperatureCelsius) /
            (MAX_BATTERY_TEMPERATURE - OPTIMAL_TEMPERATURE_MAX);
    }
}
```

# CODE SMELL 2: ANALYSE

- Fast identischer Code in beiden Klassen (DC & AC Charging Calculators)
- Nur minimale Unterschiede in den Konstanten (0.5 vs 0.7)
- Verletzt das DRY-Prinzip (Don't Repeat Yourself)
- Schwer zu warten bei Änderungen
- Fehleranfällig bei Updates

# CODE SMELL 2: LÖSUNGSVORSCHLAG

*Logik kombinieren durch Berechnung*

```
public class TemperatureEfficiencyCalculator {  
    private final double minEfficiency;  
    private final double maxEfficiency;  
  
    public TemperatureEfficiencyCalculator(double minEfficiency, double maxEfficiency) {  
        this.minEfficiency = minEfficiency;  
        this.maxEfficiency = maxEfficiency;  
    }  
  
    public double calculateEfficiency(double temperature) {  
        if (temperature < MIN_BATTERY_TEMPERATURE ||  
            temperature > MAX_BATTERY_TEMPERATURE) {  
            return 0.0;  
        }  
    }  
}
```

# REFACTORINGS (2P)

*2 unterschiedliche Refactorings aus der Vorlesung jeweils benennen, anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen  
– die Refactorings dürfen sich nicht mit den Beispielen der Code Smells überschneiden*

# REFACTORING 1: CODE

Extrahieren der Parameter-Erfassung in eine separate Methode (commit a5638e0)

## VORHER

```
lc boolean processRangeCalculation() {  
    try {  
        outputPort.showRangeCalculationHeader();  
  
        TerrainType terrain = getTerrainType();  
        WeatherType weather = getWeatherType();  
        double temperature = getTemperature();  
        DrivingEnvironment environment = getDrivingE  
        double stateOfCharge = getStateOfCharge();  
        EfficiencyMode efficiencyMode = getEfficienc  
  
        return calculateRangeUseCase.calculateRange(  
            terrain,  
            weather,  
            temperature,  
            environment.
```

## NACHHER

```
public boolean processRangeCalculation() {  
    try {  
        outputPort.showRangeCalculationHeader();  
        RangeParameters parameters = collectRangeParameters();  
        return calculateRangeUseCase.calculateRange(parameters);  
    } catch (Exception e) {  
        outputPort.showError("An error occurred while calculating the range.");  
        return false;  
    }  
}  
  
private RangeParameters collectRangeParameters() {  
    TerrainType terrain = getTerrainType();  
    WeatherType weather = getWeatherType();  
    double temperature = getTemperature();  
    DrivingEnvironment environment = getDrivingEnvironme
```

# REFACTORING 1: ERKLÄRUNG

## VORTEILE

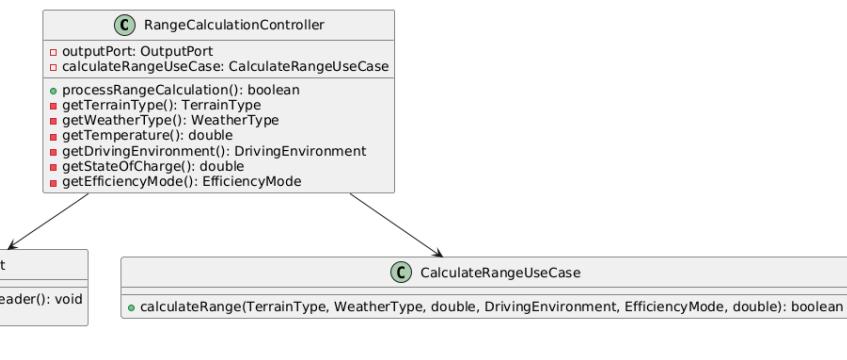
- Bessere Wartbarkeit durch modularen Code
- Einfachere Testbarkeit durch isolierte Funktionalität
- Reduzierte Komplexität in der Hauptmethode
- Einhaltung des **Single Responsibility Principle (SRP)** durch klare Trennung der Verantwortlichkeiten
- Einhaltung des **Don't Repeat Yourself (DRY)** Prinzips durch Vermeidung von Code-Duplikation

## ÄNDERUNGEN

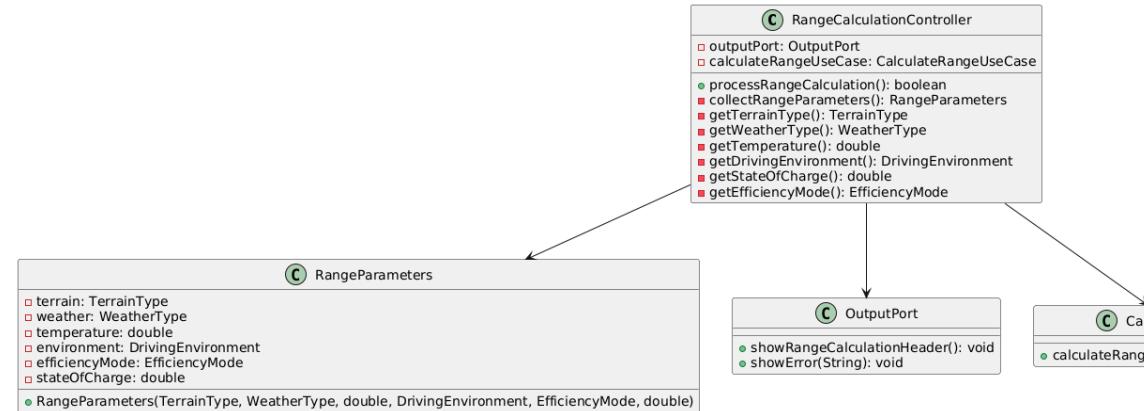
- Extraktion der Parameter-Erfassung in separate Methode  
`collectRangeParameters()`
- Einführung einer neuen Klasse `RangeParameters` zur Kapselung der Parameter
- Vereinfachung der Hauptmethode  
`processRangeCalculation()`
- Reduzierung der Parameteranzahl im Use Case Interface
- Verbesserte Lesbarkeit durch klare Trennung von Parameter-Erfassung und Geschäftslogik

# REFACTORING 1: UML-DIAGRAMME

# VORHER



# NACHHER



# REFACTORING 2: CODE

*Ersetzen der Logik in den ChargingCalculators durch Polymorphie (943b4c7)*

## VORHER

```
lc class DcChargingCalculator {  
    private double calculatePowerReductionPercent() {  
        if (targetSocPercent > 80) {  
            return 60.0;  
        } else if (targetSocPercent > 60) {  
            return 30.0;  
        } else {  
            return 5.0;  
        }  
    }  
  
    public class AcChargingCalculator {  
        private double calculateEfficiencyLoss(int connectorType) {  
            switch (connectorType) {  
                case HOUSEHOLD SOCKET:  
                    return 0.1;  
                case CAMPING SOCKET:  
                    return 0.07;  
            }  
        }  
    }  
}
```

## NACHHER

```
public interface ChargingStrategy {  
    double calculateEfficiency();  
}  
  
public class HouseholdSocketStrategy implements ChargingStrategy {  
    @Override  
    public double calculateEfficiency() {  
        return 0.1;  
    }  
}  
  
public class CampingSocketStrategy implements ChargingStrategy {  
    @Override  
    public double calculateEfficiency() {  
        return 0.07;  
    }  
}
```

# REFACTORING 2: ERKLÄRUNG

## VORTEILE

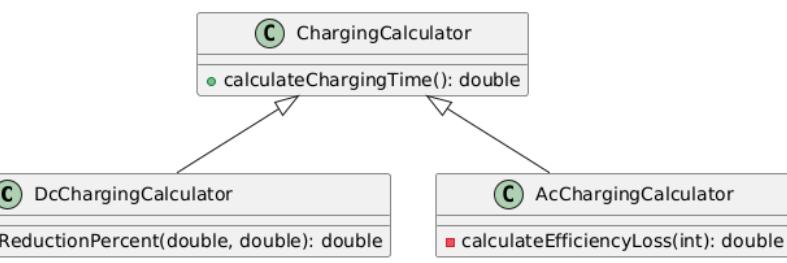
- Bessere Wartbarkeit durch klare Trennung der Verantwortlichkeiten
- Einfache Erweiterbarkeit durch neue Strategien
- Bessere Testbarkeit durch isolierte Komponenten
- Einhaltung des **Open/Closed Principle**
- Reduzierte Komplexität in den Charging Calculators

## ÄNDERUNGEN

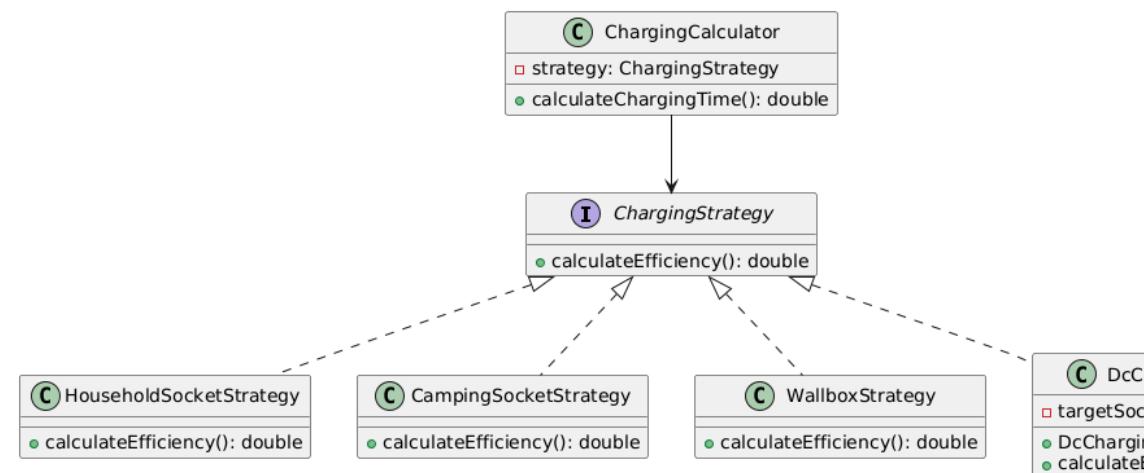
- Einführung des `ChargingStrategy` Interfaces
- Implementierung konkreter Strategien:  
`HouseholdSocketStrategy`,  
`CampingSocketStrategy`, `WallboxStrategy`,  
`DcChargingStrategy`
- Ersetzung der Logik durch polymorphe Methodenaufrufe
- Kapselung der Effizienzberechnung in den jeweiligen Strategieklassen
- Einführung von Konstruktoren für kontextabhängige Strategien

# REFACTORING 2: UML-DIAGRAMME

VORHER



NACHHER



# KAPITEL 8: ENTWURFSMUSTER (8P)

*Zwei unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils benennen, sinnvoll einsetzen, begründen und UML-Diagramm*

# ENTWURFSMUSTER 1: STRATEGY PATTERN (4P)

## ÜBERBLICK

- **Kategorie/Art:** Behavioral Pattern
- **Zweck:** Definiert eine Familie von Algorithmen, kapselt sie ab und macht sie austauschbar (Laufzeit u. Compilezeit)
- **Verwendung:** Implementierung verschiedener Berechnungsstrategien für die Reichweite von Elektrofahrzeugen

## VORTEILE

- Kapselung von unterschiedlichen Algorithmen in getrennten Klassen
- Laufzeitentscheidung über zu verwendende Strategie
- Leichte Erweiterbarkeit um neue Strategien (Auch gut für OCP)
- Einhaltung des Open/Closed Principles
- Vermeidung von bedingten Verzweigungen durch Polymorphie

# ENTWURFSMUSTER 1: STRATEGY PATTERN (FORSETZUNG)

## NACHTEILE

- Erhöhte Komplexität durch viele kleine Klassen
- Client/Klasse muss die verschiedenen Strategien kennen
- Möglicher Overhead durch zusätzliche Erstellung von Objekten
- Strategie-Auswahl kann komplex werden
- Schwieriger zu debuggen durch verteilte Logik

# STRATEGY PATTERN: INTERFACE

```
// Das Strategy Interface
public interface RangeCalculationStrategyInterface {
    RangeResult calculateRange(CarProfile carProfile, RangeParameters parameters);

    String getName();

    String getDescription();
}
```

# STRATEGY PATTERN: ERSTE IMPLEMENTIERUNG

```
// Erste Strategie-Implementierung
public class WltpBasedRangeCalculationStrategy implements RangeCalculationStrategyInterface {

    @Override
    public RangeResult calculateRange(CarProfile carProfile, RangeParameters parameters) {
        Objects.requireNonNull(carProfile, "Car profile cannot be null!!!!");
        Objects.requireNonNull(parameters, "Range parameters cannot be null");

        double baseWltpRange = carProfile.getWltpRangeKm();
        double batteryCapacity = carProfile.getBatteryProfile().getRemainingCapacityKwh();
        double currentSoC = parameters.getStateOfChargePercent();

        double baseConsumption = (batteryCapacity * 100.0) / baseWltpRange;

        double modeConsumption = baseConsumption * parameters.getEfficiencyMode().getConsumptionFactor()

        double terrainFactor = calculateTerrainFactor(parameters.getTerrain());
        double terrainConsumption = modeConsumption * terrainFactor;
    }
}
```

# STRATEGY PATTERN: ZWEITE IMPLEMENTIERUNG

```
// Zweite Strategie-Implementierung

public class ConsumptionBasedRangeCalculationStrategy implements RangeCalculationStrategyInterface {

    @Override
    public RangeResult calculateRange(CarProfile carProfile, RangeParameters parameters) {
        Objects.requireNonNull(carProfile, "Car profile cannot be null");
        Objects.requireNonNull(parameters, "Range parameters cannot be null");

        ConsumptionProfile consumptionProfile = carProfile.getConsumptionProfile();
        double baseConsumption = getBaseConsumption(consumptionProfile, parameters.getEnvironment());

        double modeConsumption = baseConsumption * parameters.getEfficiencyMode().getConsumptionFacto
            ...
        double terrainFactor = calculateTerrainFactor(parameters.getTerrain());
        double terrainConsumption = modeConsumption * terrainFactor;

        double weatherFactor = calculateWeatherFactor(parameters.getWeather(), parameters.getTempora
            ...
    }
}
```

# STRATEGY PATTERN: VERWENDUNG IM CLIENT

```
// Client-Klasse, die die Strategien verwendet
public class RangeCalculatorService {
    private final List strategies;
    private RangeCalculationStrategyInterface defaultStrategy;

    public RangeCalculatorService() {
        this.strategies = new ArrayList<>();

        WltpBasedRangeCalculationStrategy wltpStrategy = new WltpBasedRangeCalculationStrategy();
        ConsumptionBasedRangeCalculationStrategy consumptionStrategy = new ConsumptionBasedRangeCalculationStrategy();

        this.strategies.add(wltpStrategy);
        this.strategies.add(consumptionStrategy);

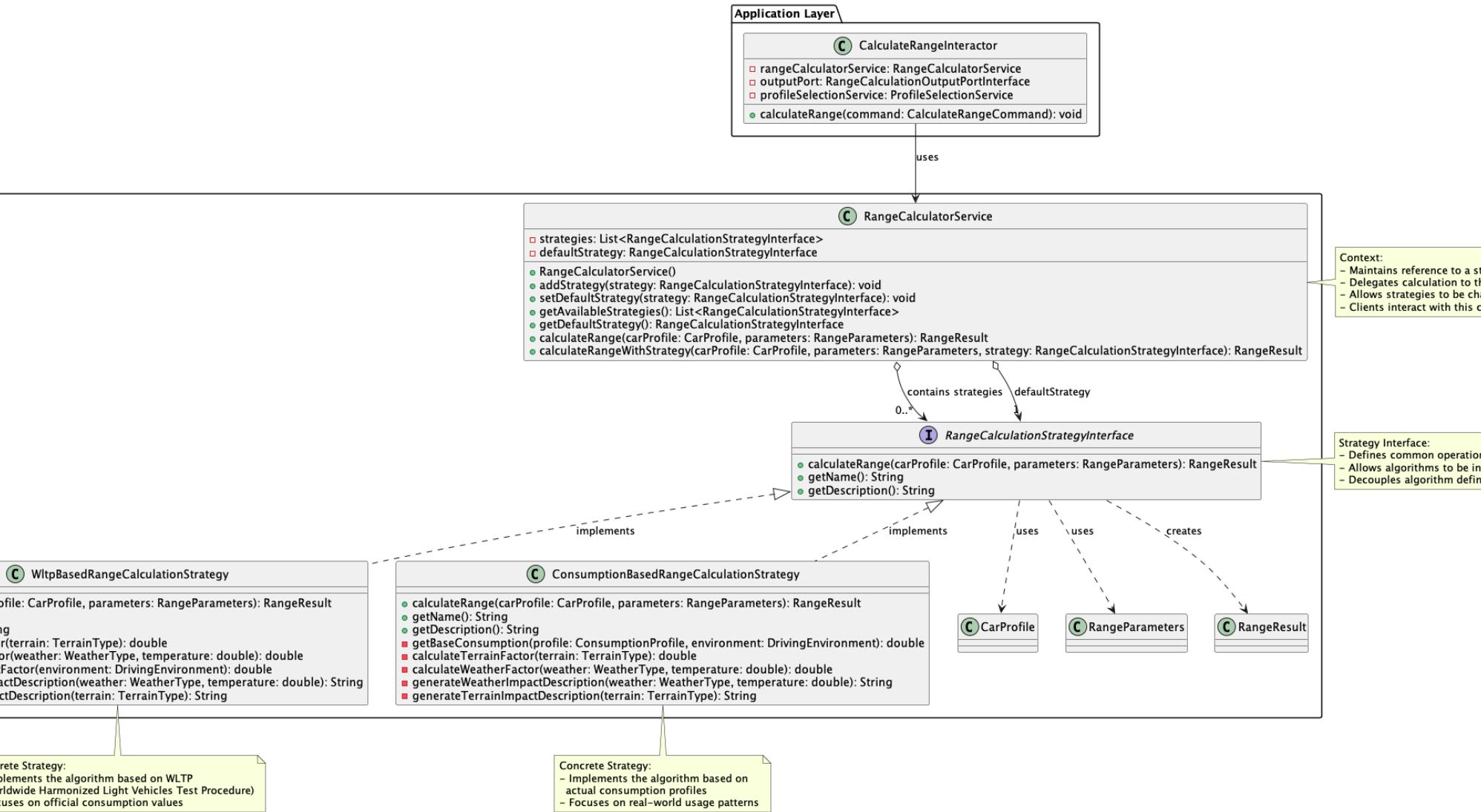
        this.defaultStrategy = consumptionStrategy;
    }

    public void addStrategy(RangeCalculationStrategyInterface strategy) {
        this.strategies.add(strategy);
    }
}
```

# STRATEGIE PATTERN: VERGLEICH

Aspekt	ConsumptionBasedRangeCalculationStrategy	WltpBasedRangeCalculationStrategy
Basis für Berechnung	Verbrauchsprofile des Fahrzeugs bei verschiedenen Geschwindigkeiten (50, 100, 130 km/h)	WLTP-Reichweite des Fahrzeugs als Fixwert
Kalkulation Grundverbrauch	Basierend auf dem aktuellen Fahrumfeld (DrivingEnvironment) wird ein Verbrauchswert (kWh/100km) gewählt	Es wird ein durchschnittlicher Verbrauch aus der WLTP-Reichweite und Batteriekapazität berechnet
Berücksichtigte Faktoren	Effizienzmodus, Geländeart, Wetterbedingungen, Temperatur, Batterieladestand	Effizienzmodus, Geländeart, Wetterbedingungen, Temperatur, Batterieladestand, zusätzlich ein Fahrumgebungs faktor (EnvironmentFactor)

# STRATEGY PATTERN: UML-DIAGRAMM



# ENTWURFSMUSTER 2: ADAPTER PATTERN (4P)

## ÜBERBLICK

- **Art/Kategorie:** Strukturmuster (Structural Pattern)
- **Zweck:** Konvertiert die Schnittstelle einer Klasse in eine andere, die vom Client erwartet wird
- **Verwendung:** Integration von I/O-Operationen in die Clean Architecture

## VORTEILE

- Flexibilität durch lose Kopplung
- Einfach zu erweitern
- Trennt Geschäftslogik von externen Ressourcen u. Komponenten
- Ermöglicht Zusammenarbeit nicht-kompatibler Schnittstellen

# **ENTWURFSMUSTER 2: ADAPTER PATTERN (FORTSETZUNG)**

## **NACHTEILE**

- Zusätzliche Komplexität durch zusätzliche Abstraktionsebene
- Möglicher Overhead durch zusätzliche Methodenaufrufe
- Erhöhte Anzahl von Klassen und Interfaces
- Externe können abrupt aufrufen

# ADAPTER PATTERN: PORT-INTERFACES

```
// Port Interface für Benutzereingaben
public interface UserInputPortInterface {
    String readLine();
    int readInt();
    double readDouble();
    boolean readBoolean(String yesOption, String noOption);
    String readStringWithValidation(String prompt, String errorMessage, java.util.function.Predicate<String> validation);
    int readIntInRange(String prompt, int min, int max);
    double readDoubleInRange(String prompt, double min, double max);
}

// Port Interface für Benutzerausgaben
public interface UserOutputPortInterface {
    void display(String message);
    void displayLine(String message);
    void displayPrompt(String prompt);
    void displayError(String errorMessage);
    void displaySuccess(String successMessage);
}
```

# ADAPTER PATTERN: CONSOLEUSERINPUTADAPTER

```
public class ConsoleUserInputAdapter implements UserInputPortInterface {  
    private final Scanner scanner;  
  
    public ConsoleUserInputAdapter(Scanner scanner) {  
        this.scanner = Objects.requireNonNull(scanner, "Scanner cannot be null");  
    }  
  
    @Override  
    public String readLine() {  
        return scanner.nextLine();  
    }  
  
    @Override  
    public int readInt() {  
        while (true) {  
            try {  
                return Integer.parseInt(scanner.nextLine());  
            } catch (NumberFormatException e) {  
                System.out.println("Please enter a valid integer value.");  
            }  
        }  
    }  
}
```

# ADAPTER PATTERN: CONSOLEUSEROUTPUTADAPTER

```
public class ConsoleUserOutputAdapter implements UserOutputPortInterface {  
    private static final String ANSI_RESET = "\u001B[0m";  
    private static final String ANSI_RED = "\u001B[31m";  
    private static final String ANSI_GREEN = "\u001B[32m";  
    private static final String ANSI_YELLOW = "\u001B[33m";  
    private static final String ANSI_CLEAR = "\u001B[H\u001B[2J";  
  
    @Override  
    public void display(String message) {  
        System.out.print(message);  
    }  
  
    @Override  
    public void displayLine(String message) {  
        System.out.println(message);  
    }  
}
```

# ADAPTER PATTERN: VERWENDUNG IM CONTROLLER

```
public class RangeCalculationController {  
    private final UserInputPortInterface userInputPort;  
    private final UserOutputPortInterface userOutputPort;  
    private final CalculateRangeUseCaseInterface calculateRangeUseCase;  
  
    public RangeCalculationController(  
        UserInputPortInterface userInputPort,  
        UserOutputPortInterface userOutputPort,  
        CalculateRangeUseCaseInterface calculateRangeUseCase) {  
        this.userInputPort = Objects.requireNonNull(userInputPort, "UserInputPort cannot be null");  
        this.userOutputPort = Objects.requireNonNull(userOutputPort, "UserOutputPort cannot be null");  
        this.calculateRangeUseCase = Objects.requireNonNull(calculateRangeUseCase, "CalculateRangeUs  
    }  
  
    public void processRangeCalculation() {  
        try {  
            // Benutzereingaben über Port sammeln  
        } catch (Exception e) {  
            // Fehlerbehandlung  
        }  
    }  
}
```

# ADAPTER PATTERN: UML-DIAGRAMM

