## Programmentwurf


- 1. Einführung
  - 1.1 Übersicht über die Applikation
  - 1.2 Wie startet man die Applikation?
  - 1.3 Wie testet man die Applikation?
- 2. Clean Architecture
  - 2.1 Was ist Clean Architecture?
  - 2.2 Analyse der Dependency Rule
    - 2.2.1 Positiv-Beispiel: `CustomerRepositoryImpl`
    - 2.2.2 Positiv-Beispiel: `PolicyManagementImpl`
  - 2.3 Analyse der Schichten
    - 2.3.1 Schicht: Applikations-Schicht
    - 2.3.2 Schicht: Domain-Schicht
- 3. SOLID
  - 3.1 Analyse Single-Responsibility-Principle (SRP)
    - 3.1.1 Positiv-Beispiel: `BasicPremiumCalculationStrategy`
    - 3.1.2 Negativ-Beispiel: `WriteCustomerManagementImpl`
  - 3.2 Analyse Open-Closed-Principle (OCP)
    - 3.2.1 Positiv-Beispiel: `PremiumCalculationStrategyFactory`
    - 3.2.2 Negativ-Beispiel: `WriteCustomerManagementImpl`
  - 3.3 Analyse Interface-Segregation-Principle (ISP)
    - 3.3.1 Positiv-Beispiel: `ReadCustomerManagement` und `WriteCustomerManagement`
    - 3.3.2 Negativ-Beispiel: `CustomerRepository`
- 4. Weitere Prinzipien
  - 4.1 Analyse GRASP: Geringe Kopplung
    - 4.1.1 Positives-Beispiel: `PolicyManagementImpl`
    - 4.1.2 Negatives-Beispiel: `TicketManagementImplTest`
  - 4.2 Analyse GRASP: Hohe Kohäsion
  - 4.3 Don’t Repeat Yourself (DRY)
- 5. Unit Tests
  - 5.1 Zehn Unit Tests - Tabelle
  - 5.2 ATRIP
    - 5.2.1 ATRIP: Automatic
    - 5.2.2 ATRIP: Thorough
      - 5.2.2.1 Positiv-Beispiel
      - 5.2.2.2 Positiv-Beispiel

        - 5.2.3 ATRIP: Professional
            - 5.2.3.1 Positiv-Beispiel
            - 5.2.3.2 Negativ-Beispiel
    - 5.3 Code Coverage
    - 5.4 Fakes und Mocks
        - 5.4.1 Mock-Objekt: `CustomerRepository`
        - 5.4.2 Mock-Objekt: `WriteCustomerManagement`
- 6. Domain-Driven-Design (DDD)
    - 6.1 Ubiquitous Language
        - 6.1.1 Entities
        - 6.1.2 Nutzer
        - 6.1.3 Tabelle
    - 6.2 Entities - Policy Entity
    - 6.3 Value Objects - Premium Value Object
    - 6.4 Aggregates - Customer Aggregate
    - 6.5 Repositories - Customer Repository
- 7. Refactoring
    - 7.1 Code Smells
        - 7.1.1 Long Method
        - 7.1.2 Duplicated Code
    - 7.2 Refactorings
        - 7.2.1 Replace Conditional with Polymorphism
        - 7.2.2 Extract Method
- 8. Design Patterns
    - 8.1 Strategy Pattern
    - 8.2 Builder Pattern


# 1. Einführung

## 1.1 Übersicht über die Applikation

Die Applikation SRI (*Simon Stefan Insuranci*) ist eine Software zur Verwaltung von Autoversicherungen. Sie ermöglicht es, VersicherungsPolicies für Kunden zu erstellen und zu verwalten. Die Applikation berechnet die Versicherungskosten basierend auf verschiedenen Faktoren wie dem Wert des Autos, dem Alter des Kunden, Verkehrsverstößen wie Tickets und Unfällen.

**Funktionsweise:**

1. **Kostenberechnung:** Die Applikation verwendet verschiedene Strategien zur Berechnung der Versicherungsprämien (z.B. Basic, Standard, Deluxe).
2. **Regeln:** Es gibt spezifische Regeln, die die Prämien beeinflussen, wie zusätzliche Gebühren für junge oder ältere Fahrer, erhöhte Prämien bei Verkehrsverstößen und Unfällen, sowie Ausschlusskriterien für sehr teure Autos.
3. **Verwaltung:** Mitarbeiter der Versicherungsfirma können Kunden und deren Policies verwalten, Unfälle und Tickets hinzufügen und die Auswirkungen auf die Prämien sehen.

**Zweck:**
Die Applikation soll den Prozess der Verwaltung von Autoversicherungen effizienter und transparenter gestalten, indem sie automatisierte Berechnungen und klare Regeln zur Prämienbestimmung bietet.

### 1.2 Wie startet man die Applikation?

**Voraussetzungen:**

- Java Development Kit (JDK) Version 21
- Apache Maven Version 3.9.9

**Schritt-für-Schritt-Anleitung:**

1. **Repository klonen:**

```
git clone https://github.com/SirSimon04/clean-car-insurance
cd clean-car-insurance
```

2. **Projekt bauen:**

```
mvn clean install
```

3. **Applikation starten:**

```
cd 0-insurance-main
mvn exec:java -Dexec.mainClass="de.sri.Main"
```

4. **Applikation über die Konsole verwenden:**

“/assets/console_output.png” could not be found.

### 1.3 Wie testet man die Applikation?

```
cd clean-car-insurance
mvn test
```

Die Testergebnisse werden im Terminal angezeigt.

“/assets/console_test_ergebnisse.png” could not be found.

## 2. Clean Architecture

### 2.1 Was ist Clean Architecture?

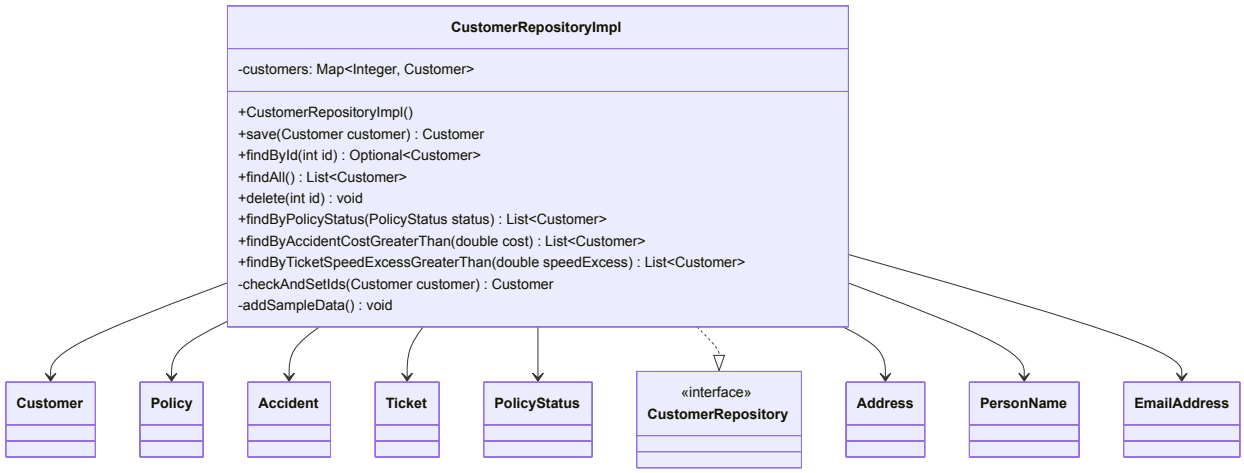
Clean Architecture ist ein Architekturstil für Software, der darauf abzielt, die Abhängigkeiten zwischen den verschiedenen Komponenten einer Anwendung zu minimieren und die Testbarkeit, Wartbarkeit und Flexibilität zu erhöhen. Die Hauptidee ist, dass die Geschäftslogik (Use Cases) unabhängig von Frameworks, Datenbanken, UI oder anderen externen Systemen bleibt. Dies wird durch die strikte Trennung der Verantwortlichkeiten und die Einhaltung der Dependency Rule erreicht.

### 2.2 Analyse der Dependency Rule

Aufgrund der Projektstruktur wird die Dependency Rule der Clean Architecture immer befolgt und kann nicht dagegen verstoßen werden. Deswegen werden zwei positive Beispiele aufgeführt.

```
├── 0-insurance-main
├── 1-insurance-adapters
├── 2-insurance-application
├── 3-insurance-domain
├── README.md
└── pom.xml
```

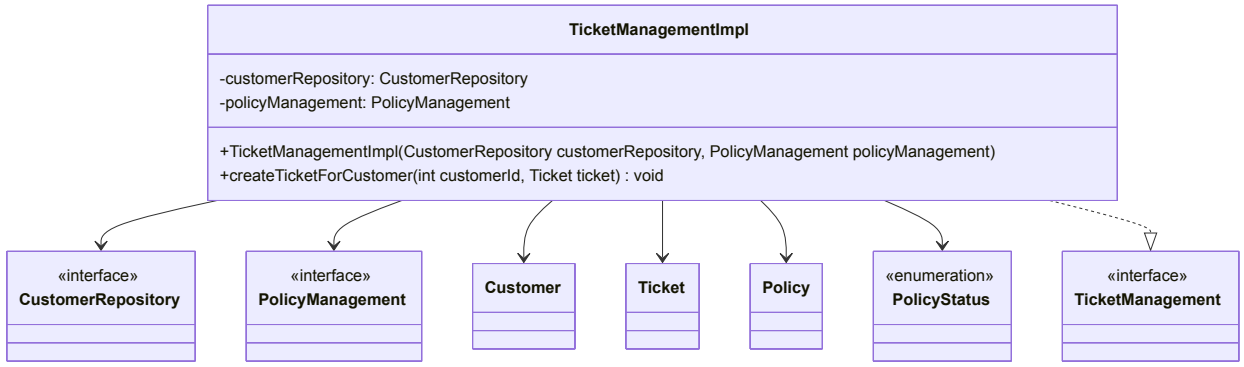
#### 2.2.1 Positiv-Beispiel: `CustomerRepositoryImpl`




**Analyse:**

- **Abhängigkeiten:** `CustomerRepositoryImpl` hängt von dem Interfaces `CustomerRepository` und sämtlichen Entity Klassen ab.
- **Einhaltung der Dependency Rule:** Die Klasse `CustomerRepositoryImpl` befindet sich in der Applikations-Schicht und die abhängenden Entitäten befinden sich in der Domain-Schicht. Dies entspricht der Dependency Rule, da die Abhängigkeiten von außen nach innen verlaufen und nicht umgekehrt.

### 2.2.2 Positiv-Beispiel: `PolicyManagementImpl`



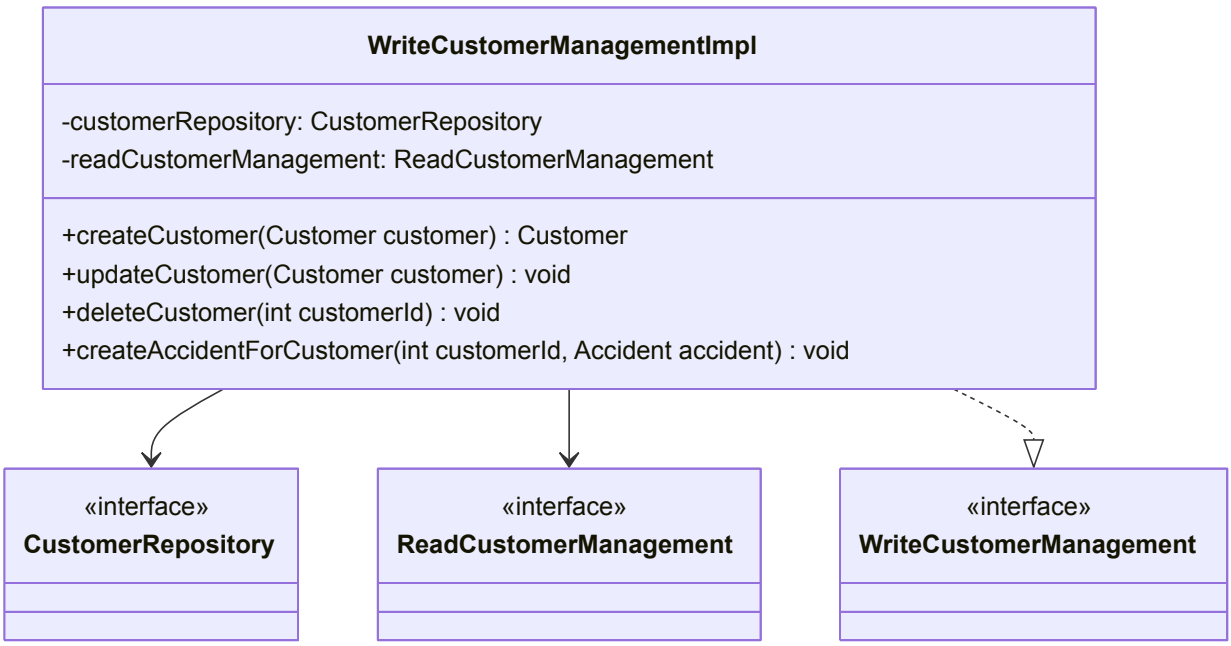

**Analyse:**

- **Abhängigkeiten:** `TicketManagementImpl` implementiert das Interface `TicketManagement` und hängt von den Interfaces `CustomerRepository` und `PolicyManagement` sowie weiteren Entity-Klassen ab.
- **Einhaltung der Dependency Rule:** Die Klasse `TicketManagementImpl` befindet sich in der Applikations-Schicht und die Interfaces sowie die abhängenden Entitäten befinden sich in der Domain-Schicht. Dies entspricht der Dependency Rule, da die Abhängigkeiten von außen nach innen verlaufen und nicht umgekehrt.

## 2.3 Analyse der Schichten

### 2.3.1 Schicht: Applikations-Schicht

**Klasse:** `CustomerManagementUseCase`




**Beschreibung der Aufgabe:**

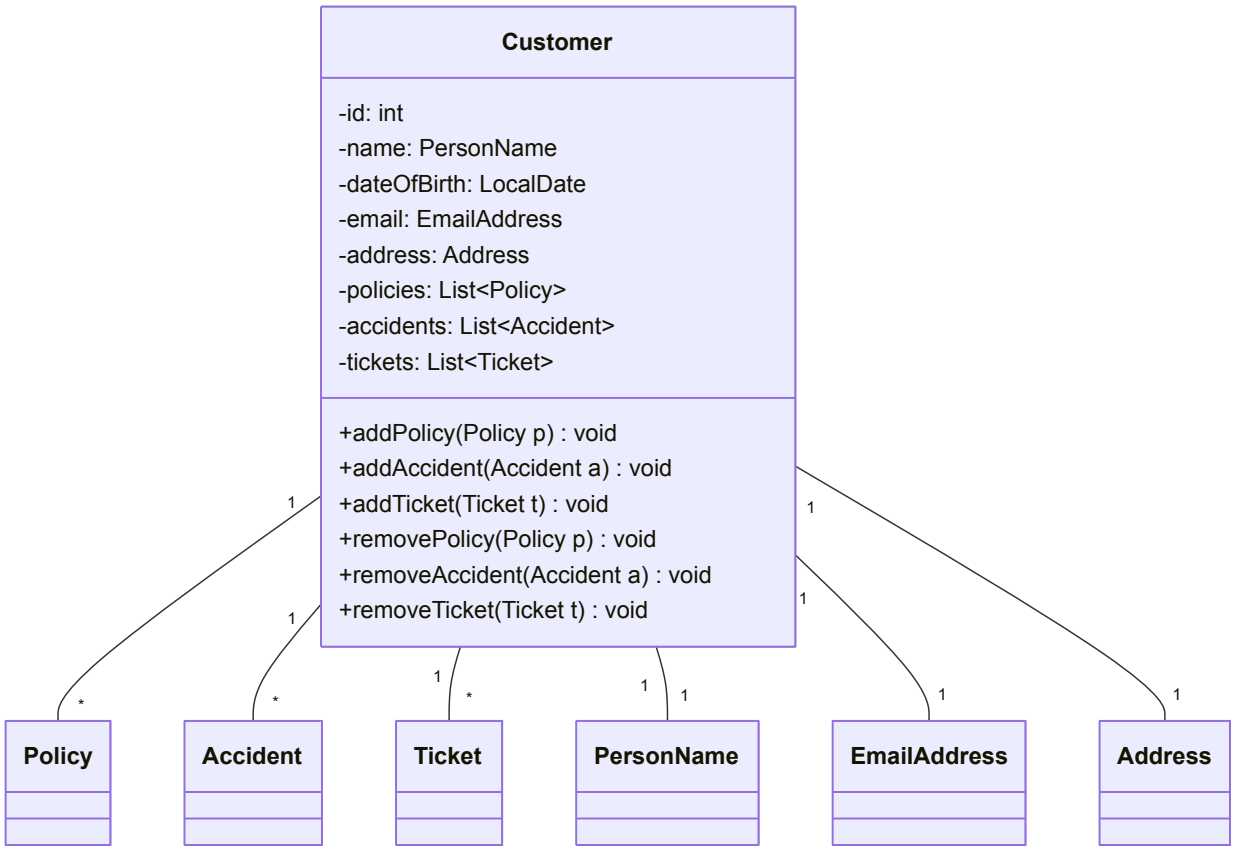
Die Klasse `WriteCustomerManagementImpl` ist verantwortlich für die schreibende Verwaltung der Kunden. Sie bietet Methoden zum Hinzufügen, Entfernen und Aktualisieren von Kunden. Sie interagiert mit dem `CustomerRepository`, um die Datenpersistenz zu gewährleisten.

**Einordnung in die Clean-Architecture:**

Die Klasse gehört zu der Applikations-Schicht, da sie die Geschäftslogik für die Verwaltung der Kunden kapselt. Sie stellt sicher, dass die Geschäftslogik unabhängig von der Datenpersistenz bleibt und nur über Abstraktionen (Interfaces) mit der Datenbank interagiert. Die Interfaces gehören alle zur Domain-Schicht.

### 2.3.2 Schicht: Domain-Schicht

**Klasse:** `Customer`




**Beschreibung der Aufgabe:**

Die Klasse `Customer` repräsentiert einen Kunden der Autoversicherung. Sie enthält alle relevanten Informationen über den Kunden, wie Name, Geburtsdatum, Adresse und die Liste der Policies, Accidents und Tickets.

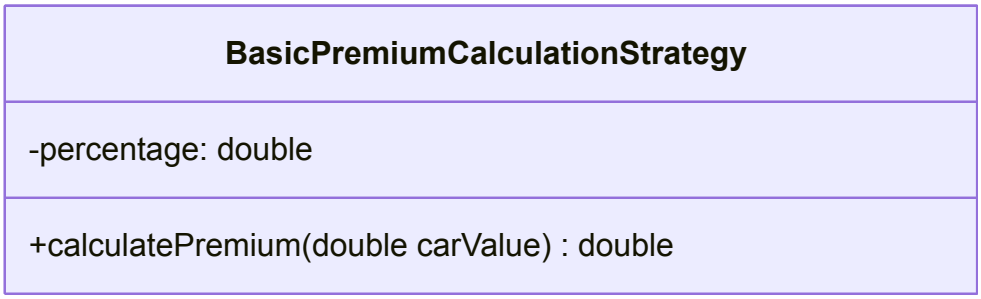
**Einordnung in die Clean-Architecture:**

Die Klasse gehört zur Domain-Schicht, da sie eine zentrale Rolle in der Domäne der Applikation spielt und die wesentlichen Daten eines Kunden kapselt. Sie ist unabhängig von anderen Schichten und kann in verschiedenen Klassen der Applikations-Schicht konsumiert werden.

## 3. SOLID

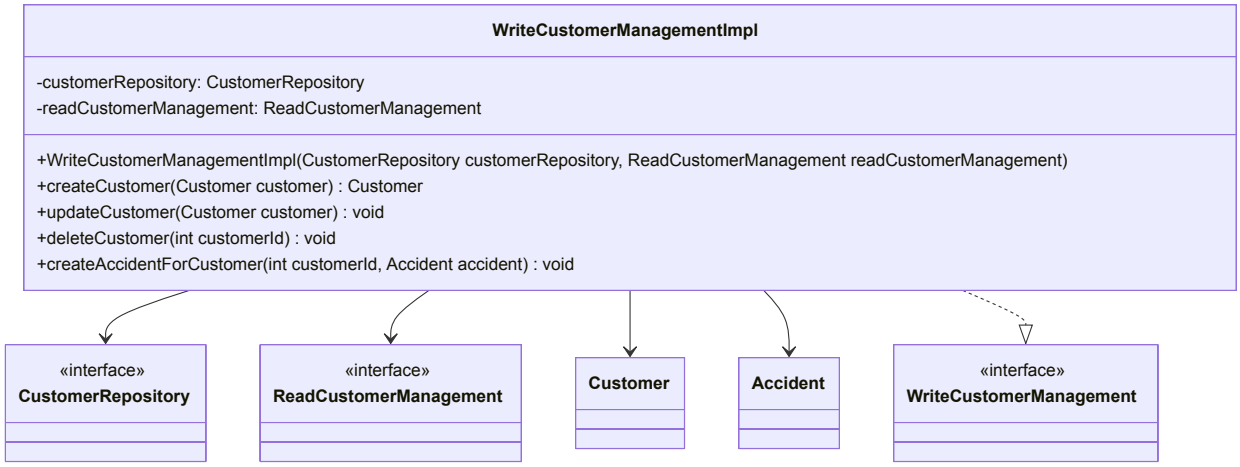
### 3.1 Analyse Single-Responsibility-Principle (SRP)

#### 3.1.1 Positiv-Beispiel: `BasicPremiumCalculationStrategy`



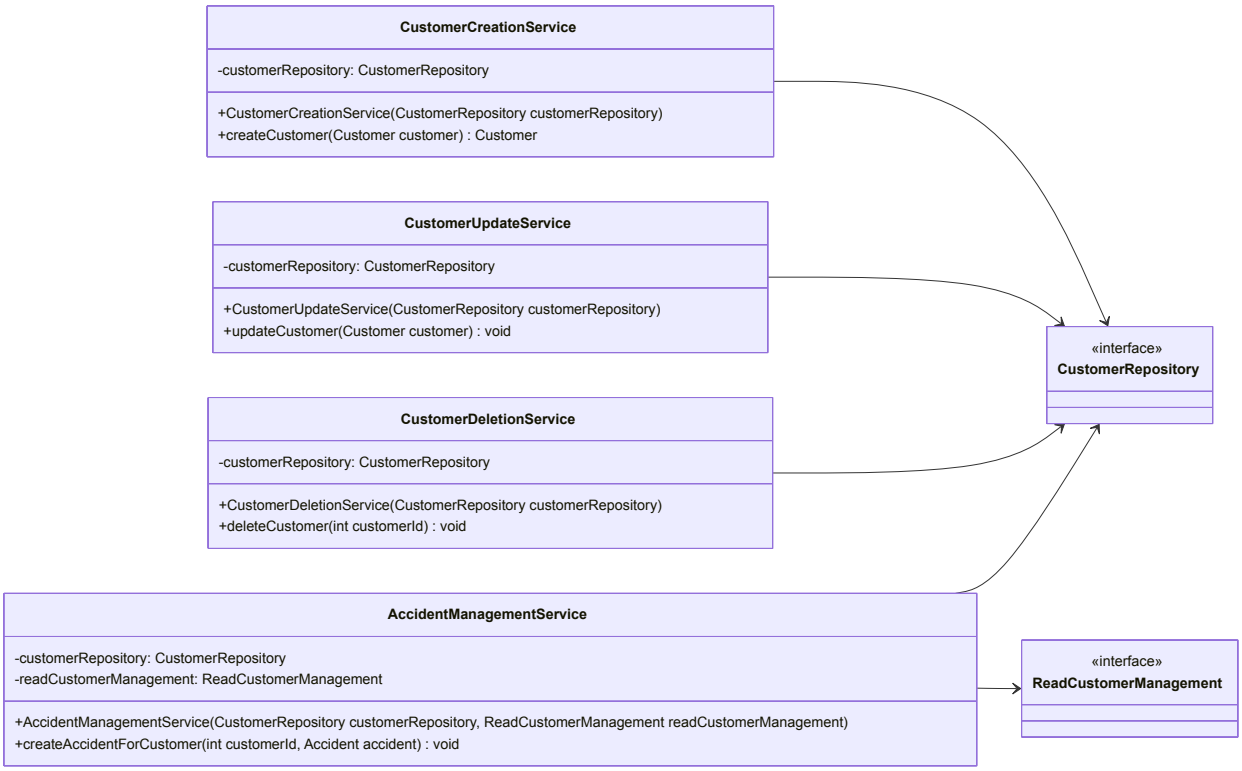

**Beschreibung der Aufgabe:**
Die Klasse `BasicPremiumCalculationStrategy` hat nur eine einzige Verantwortung: die Berechnung des Premiums basierend auf einem festen Prozentsatz des Autowertes. Sie erfüllt das SRP, da sie nur eine Aufgabe hat und diese klar definiert ist.

### 3.1.2 Negativ-Beispiel: `WriteCustomerManagementImpl`



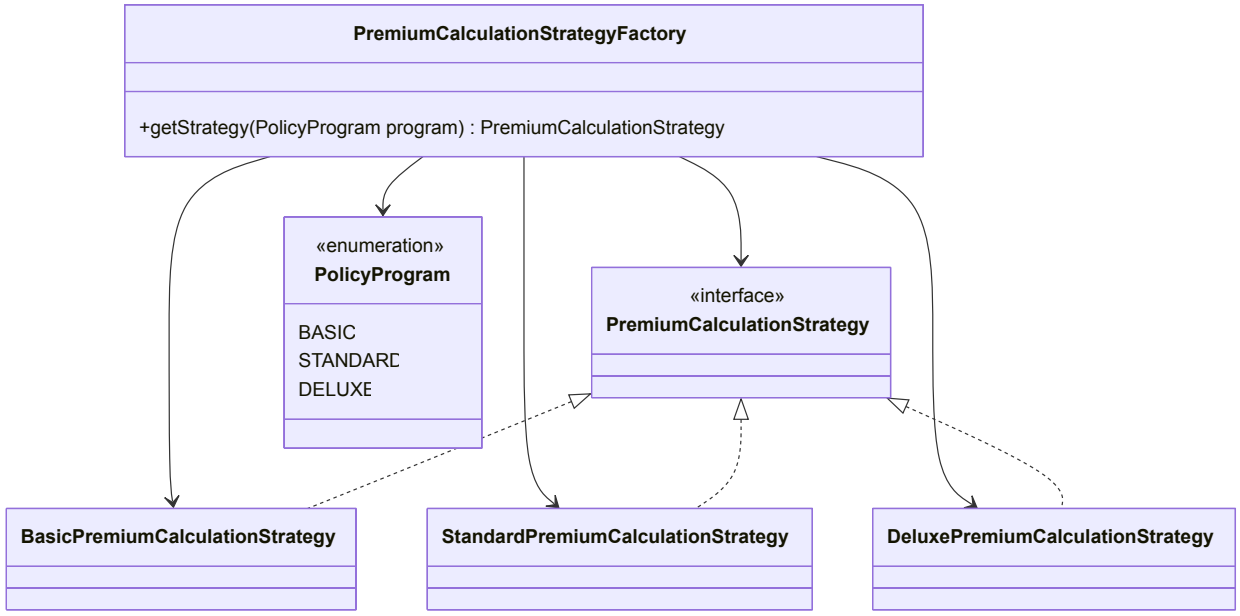

**Beschreibung der Aufgaben:**
Die Klasse `WriteCustomerManagementImpl` hat mehrere Verantwortlichkeiten: das Erstellen, Aktualisieren und Löschen von Kunden sowie das Hinzufügen von Unfällen zu Kunden. Dies verletzt das SRP, da die Klasse mehrere Aufgaben hat.

**Möglicher Lösungsweg:**
Aufteilung der Klasse `WriteCustomerManagementImpl` in einen `CustomerCreationService`, `CustomerUpdateService`, `CustomerDeletionService` und `AccidentManagementService`. Dadurch werden die mehreren Verantwortlichkeiten in mehrere einzelnen Klassen gekapselt.



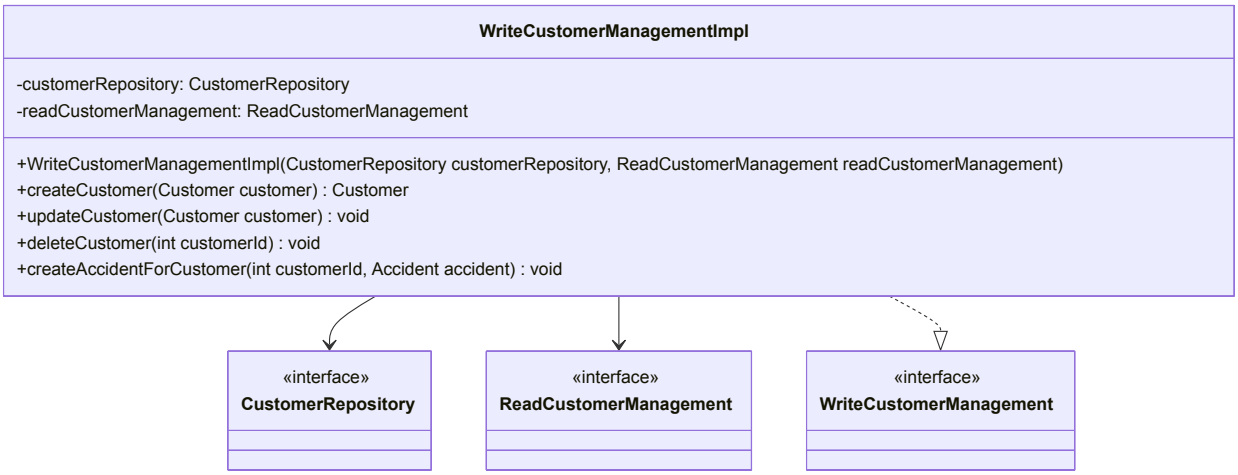

## 3.2 Analyse Open-Closed-Principle (OCP)

### 3.2.1 Positiv-Beispiel: `PremiumCalculationStrategyFactory`



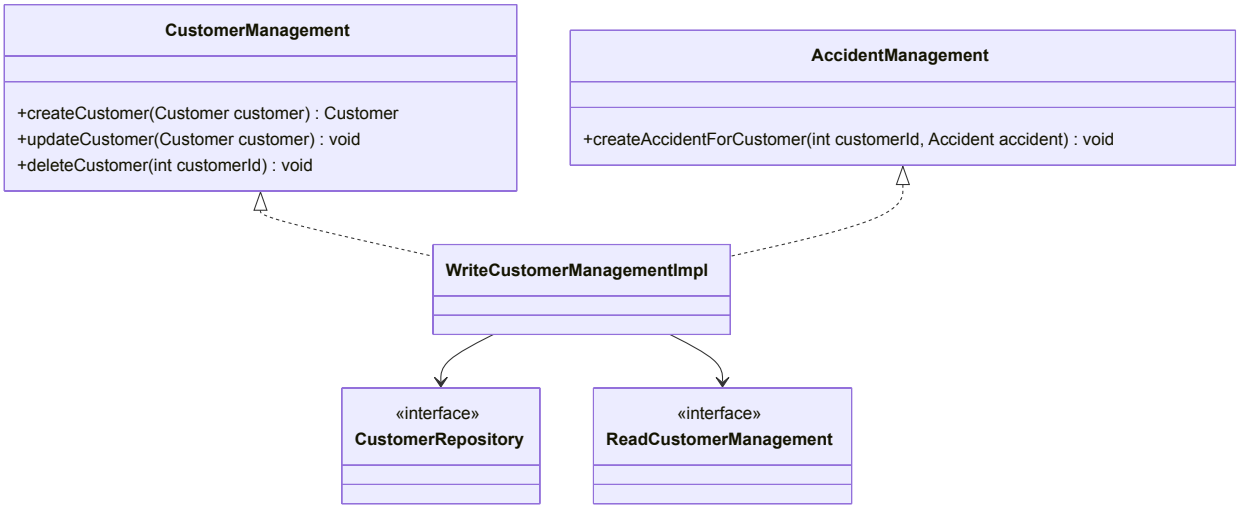

**Analyse:**
Die Klasse `PremiumCalculationStrategyFactory` ist offen für Erweiterungen, da neue Berechnungsstrategien für das Premium hinzugefügt werden können, ohne die bestehende Klasse zu ändern. Dies wird durch die Verwendung eines Enums und eines Switch-Statements erreicht, das leicht erweitert werden kann.

### 3.2.2 Negativ-Beispiel: `WriteCustomerManagementImpl`




**Analyse:**
Die Klasse `WriteCustomerManagementImpl` ist nicht offen für Erweiterungen, da jede neue Funktionalität (z.B. das Hinzufügen neuer Methoden) Änderungen an der bestehenden Klasse erfordert. Dies verletzt das OCP.

**Möglicher Lösungsweg:**

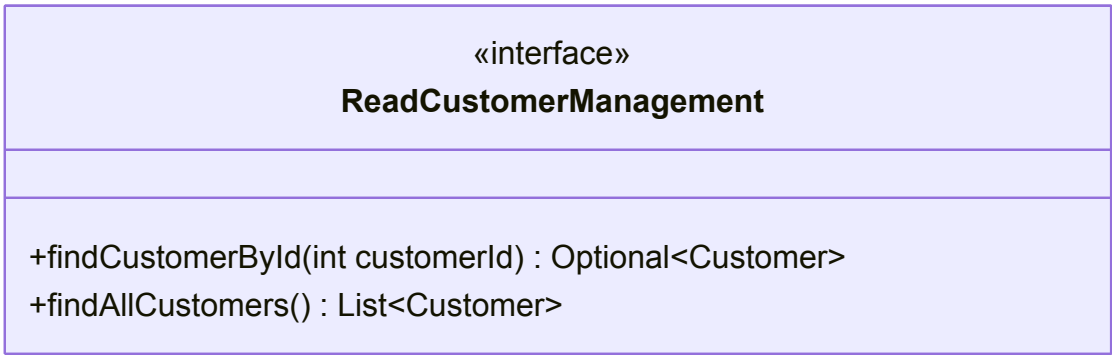



## 3.3 Analyse Interface-Segregation-Principle (ISP)

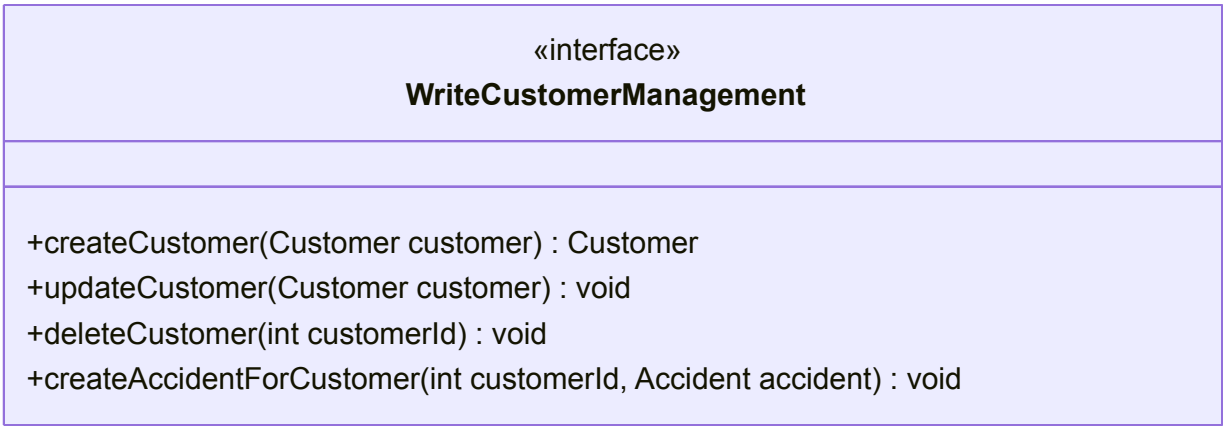
### 3.3.1 Positiv-Beispiel: `ReadCustomerManagement` und `WriteCustomerManagement`

**Analyse:**
Die Interfaces `ReadCustomerManagement` und `WriteCustomerManagement` sind nach Funktionalität (Lese- und Schreibzugriff) aufgeteilt. Dies erfüllt das ISP, da die Implementierungen nicht gezwungen sind, unnötige Methoden zu implementieren. Dadurch kann verhindert werden, dass ungewollte Schreibzugriffe ausgeführt werden.

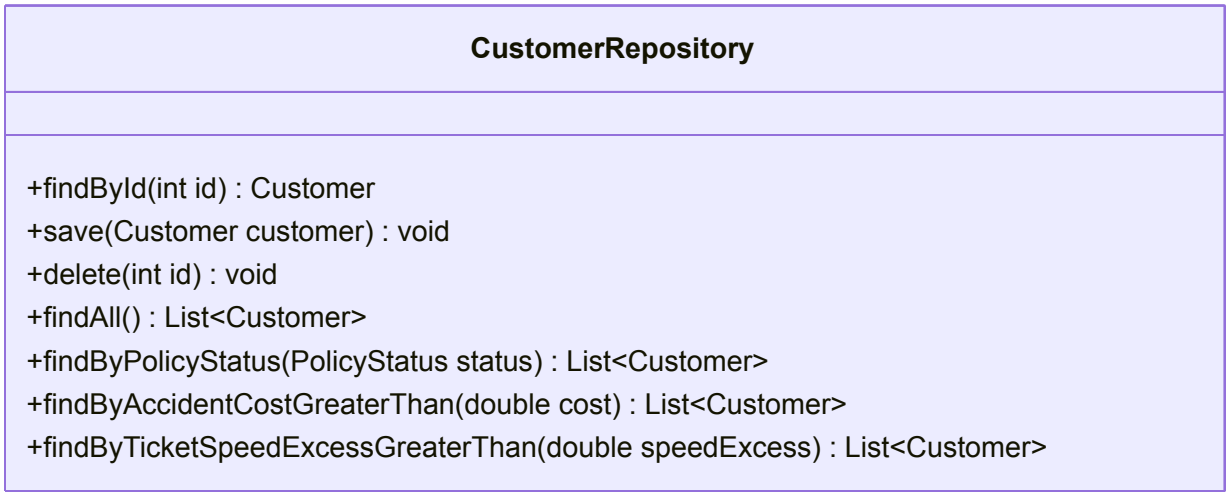
**Interface:** `ReadCustomerManagement`




**Interface:** `WriteCustomerManagement`

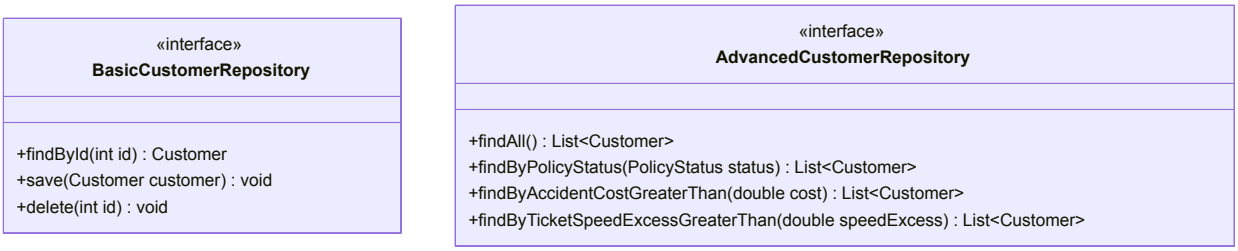



### 3.3.2 Negativ-Beispiel: `CustomerRepository`




**Analyse:**
Das Interface `CustomerRepository` hat viele Methoden, die möglicherweise nicht von allen Implementierungen benötigt werden. Dies verletzt das ISP, da Implementierungen gezwungen sind, Methoden zu implementieren, die sie nicht benötigen.

**Möglicher Lösungsweg:**

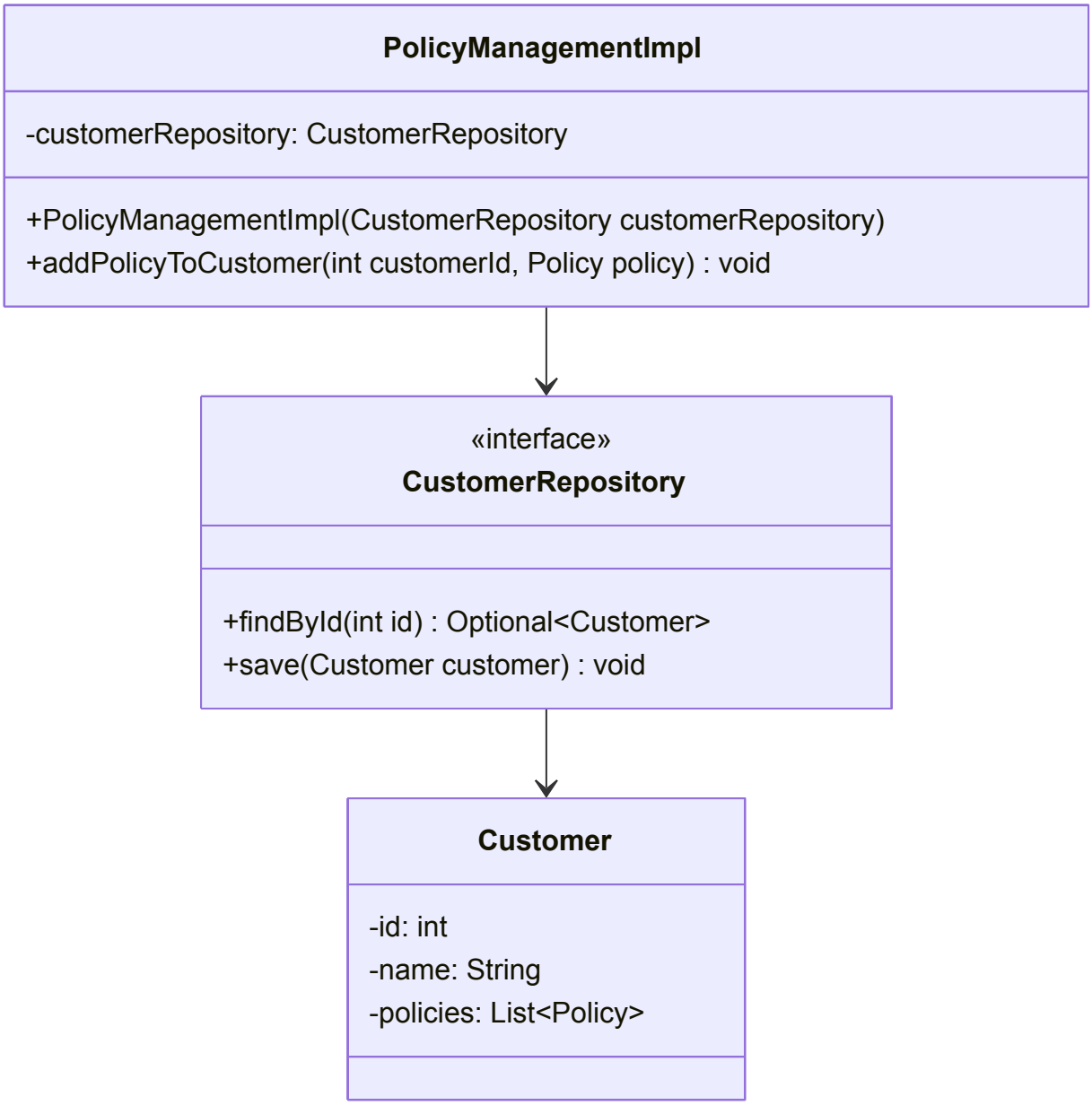



**Beschreibung:**
Durch die Aufteilung des `CustomerRepository`-Interfaces in `BasicCustomerRepository` und `AdvancedCustomerRepository` wird das ISP erfüllt, da Implementierungen nur die Methoden implementieren müssen, die sie tatsächlich benötigen.

## 4. Weitere Prinzipien

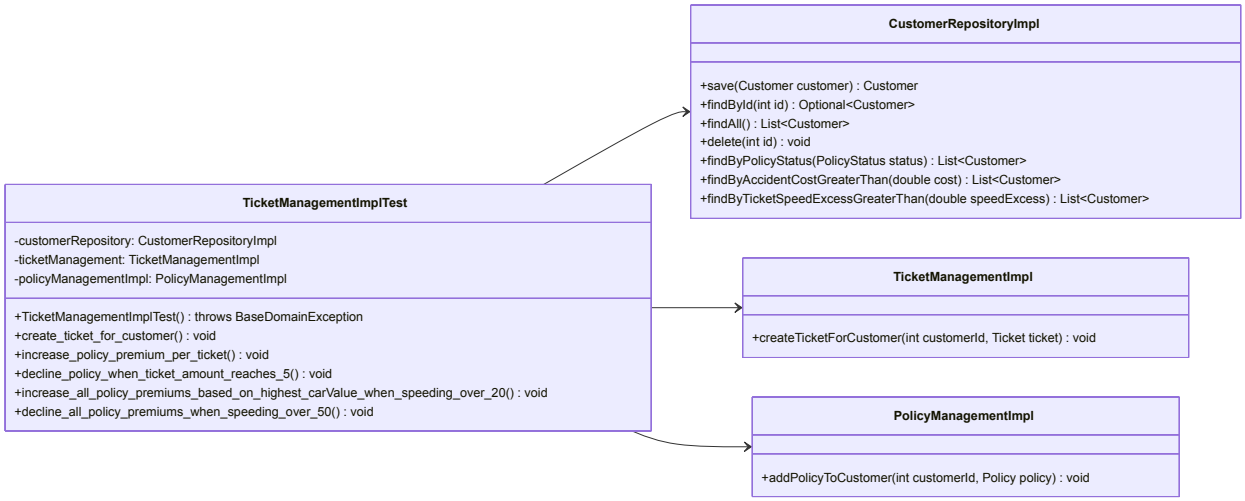
### 4.1 Analyse GRASP: Geringe Kopplung

#### 4.1.1 Positives-Beispiel: `PolicyManagementImpl`




**Analyse:**
Die Klasse `PolicyManagementImpl` weist eine geringe Kopplung auf, da sie vom Interface `CustomerRepository` und nicht von einer konkreten Implementierung abhängt. Dies ermöglicht Flexibilität und einfachere Tests, da verschiedene Implementierungen von `CustomerRepository` verwendet werden können, ohne die Klasse `PolicyManagementImpl` zu ändern. Die Abstraktion der Datenzugriffslogik durch das Interface reduziert die direkten Abhängigkeiten, wodurch das System wartbarer und anpassungsfähiger wird.

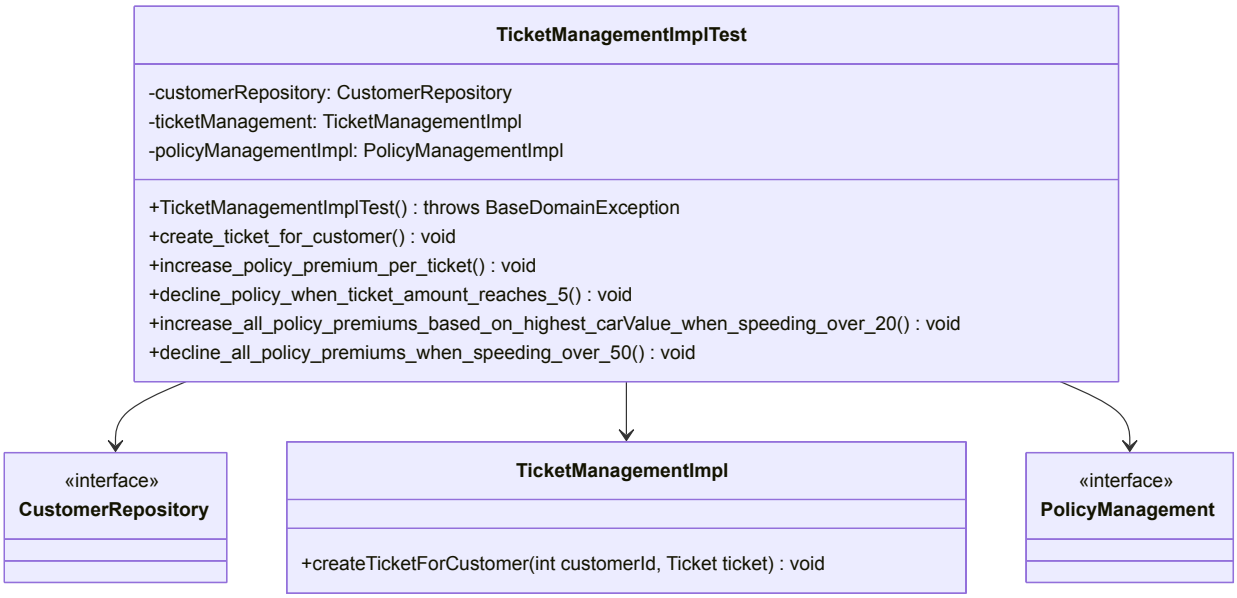
### 4.1.2 Negatives-Beispiel: `TicketManagementImplTest`




**Analyse:**
Die Klasse `TicketManagementImplTest` ist verantwortlich für das Testen der `TicketManagementImpl`-Klasse. Sie erstellt Instanzen von `CustomerRepositoryImpl`, `PolicyManagementImpl` und `TicketManagementImpl` im Konstruktor und verwendet diese in den Testmethoden. Dabei hängt sie nicht von den Interfaces `CustomerRepository` und `PolicyManagement` ab, sondern von den konkreten Implementierungen `CustomerRepositoryImpl` und `PolicyManagementImpl`. Dies führt zu einer hohen Kopplung und macht die Tests anfällig für Änderungen in den Implementierungen von `CustomerRepositoryImpl` und `PolicyManagementImpl`. Werden Änderungen an der Funktionalität an diesen beiden Komponenten vorgenommen, müssen auch die Tests in `TicketManagementImplTest` angepasst werden, wobei die Tests nicht durch Änderungen an diesen Komponeneten beeinflusst werden sollen.

**Möglicher Lösungsweg:**
Um die Kopplung zu reduzieren, sollten `CustomerRepositoryImpl` und `PolicyManagementImpl` gemockt werden.

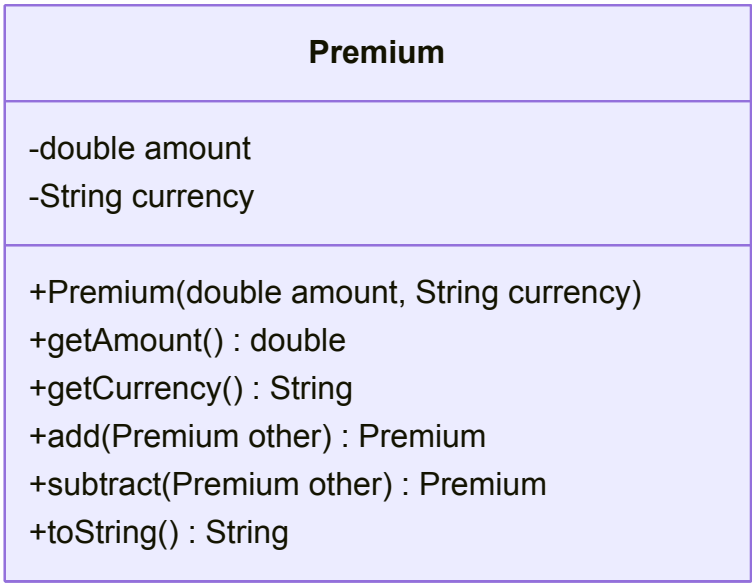
**UML Diagramm nach Refactoring:**




Durch das Mocken von `CustomerRepository` und `PolicyManagement` wird die Kopplung reduziert, da die `TicketManagementImplTest`-Klasse nun von den Interfaces `CustomerRepository` und `PolicyManagement` abhängt, anstatt von der konkreten Implementierung `CustomerRepositoryImpl`. Dies ermöglicht eine einfachere Austauschbarkeit der Implementierungen und erhöht die Flexibilität und Wartbarkeit der Tests.

### 4.2 Analyse GRASP: Hohe Kohäsion

**Klasse:** `Premium`




**Begründung:**
Die Klasse `Premium` weist eine hohe Kohäsion auf, da alle Attribute und Methoden semantisch eng miteinander verbunden sind und sich auf die Verwaltung der Kosten einer Versicherung konzentrieren. Die Attribute `amount` und `currency` beschreiben die

wesentlichen Eigenschaften der Kosten einer Versicherung. Die Methoden der Klasse (`getAmount`, `getCurrency`, `add`, `subtract`, `toString`) arbeiten direkt mit diesen Attributen und bieten eine klare und verständliche Schnittstelle zur Manipulation und Abfrage der Premium-Daten.

**Vorteile hoher Kohäsion:**
Die Klasse `Premium` hat ein einfaches und verständliches Design, da sie sich auf eine einzige Verantwortlichkeit konzentriert: die Verwaltung der Kosten einer Versicherung. Durch die klare Trennung der Verantwortlichkeiten und die enge semantische Verbindung der Attribute und Methoden kann die Klasse `Premium` in verschiedenen Kontexten wiederverwendet werden, ohne dass Änderungen erforderlich sind.

**Technische Metriken:**
Die Klasse `Premium` hat eine überschaubare Anzahl von Attributen und Methoden, was zur Übersichtlichkeit beiträgt.
Die Methoden der Klasse `Premium` nutzen die Attribute der Klasse intensiv, was auf eine hohe Kohäsion hinweist.

### 4.3 Don’t Repeat Yourself (DRY)

**Commit Hash:** 44667a8446d2a43913175c99e16068f6a90446eb

**Code-Beispiel Vorher:**

Dieses Beispiel wäre im Laufe der Entwicklung noch komplizierter geworden, da der Name und E-Mail durch eigene Value-Objects ersetzt wurden.

```
String firstName = "Anna";
String lastName = "Schmidt";
LocalDate dateOfBirth = LocalDate.of(1987, 6, 15);
String email = "anna.schmidt@example.com";
Address address = new Address("Musterstraße 1", "12345",
"Musterstadt");
Customer customer = new Customer(0, firstName, lastName, dateOfBirth,
email, address);
```

**Code-Beispiel Nachher:**

```
Customer customer = new TestCustomerDirector(new
Customer.Builder()).createMockUser()
```

**Begründung und Auswirkung:**
Durch die Verwendung des `TestCustomerDirector` zur Erstellung von `Customer`-Objekten wird duplizierte Logik vermieden und die Erstellung von `Customer`-Objekten zentralisiert. In sämtlichen Tests wurden vorher die Customer Objekte manuell angelegt, diese Erstellung mit Mitgabe der Parameter war in jedem Test zi finden. Durch das Zusammenfassen dieser Erstellung in `TestCustomerDirector.createMockuser` ist diese Erstellung von `Customer-Objekten` nur noch einmal zu finden. Dies erhöht die Wartbarkeit und Lesbarkeit des Codes, da Änderungen an der Erstellung von `Customer`-Objekten nur an einer Stelle vorgenommen werden müssen. Außerdem werden die Tests deutlich kürzer.

## 5. Unit Tests

### 5.1 Zehn Unit Tests - Tabelle

Die folgenden aufgeführten Tests befinden sich in der `PolicyManagementImplTest` Klasse.

| Unit Test | Beschreibung |
|---|---|
| `add_basic_policy` | Testet das Hinzufügen einer BASIC-Policy zu einem Kunden und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_policy_basic_with_young_driver` | Testet das Hinzufügen einer BASIC-Policy zu einem jungen Fahrer und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_policy_basic_with_senior_driver` | Testet das Hinzufügen einer BASIC-Policy zu einem älteren Fahrer und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_basic_policy_with_car_value_fee` | Testet das Hinzufügen einer BASIC-Policy mit einem hohen Autowert und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_standard_policy` | Testet das Hinzufügen einer STANDARD-Policy zu einem Kunden und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_deluxe_policy` | Testet das Hinzufügen einer DELUXE-Policy zu einem Kunden und überprüft die Prämienberechnung und die Anzahl der Policies des Kunden. |
| `add_policy_with_too_high_car_value` | Testet das Hinzufügen einer Policy mit einem zu hohen Autowert und überprüft, ob |

| Unit Test | Beschreibung |
| --- | --- |
| | eine `CarTooExpensiveException` geworfen wird. |
| `add_policy_with_customer_under_18_years_old` | Testet das Hinzufügen einer Policy zu einem Kunden unter 18 Jahren und überprüft, ob eine `CustomerTooYoungException` geworfen wird. |
| `increase_all_policies_premium` | Testet die Erhöhung der Prämien aller Policies eines Kunden und überprüft die neue Prämienhöhe. |
| `add_policy_with_customer_not_found` | Testet das Hinzufügen einer Policy zu einem nicht existierenden Kunden und überprüft, ob eine `CustomerNotFoundException` geworfen wird. |

## 5.2 ATRIP

### 5.2.1 ATRIP: Automatic

**Begründung für automatisches Testen**

JUnit-Tests werden in einem Maven-Projekt automatisch während der Test-Phase ausgeführt, weil Maven das `maven-surefire-plugin` standardmäßig verwendet. Dieses Plugin ist darauf ausgelegt, JUnit-Tests zu erkennen und auszuführen.

### 5.2.2 ATRIP: Thorough

#### 5.2.2.1 Positiv-Beispiel

**Code-Beispiel:**

```
@Test
void add_basic_policy() throws CustomerNotFoundException,
CustomerTooYoungException, CarTooExpensiveException {
    Customer customer = new TestCustomerDirector(new
Customer.Builder()).createMockUser();

when(customerRepository.findById(1)).thenReturn(Optional.of(customer))
;

    Policy policy = new Policy(1, PolicyStatus.ACTIVE,
PolicyProgram.BASIC, 10000);
    policyManagement.addPolicyToCustomer(1, policy);
```

```
    assertEquals(500, policy.getPremium().getAmount());
    assertEquals(1, customer.getPolicies().size());
    assertEquals(policy, customer.getPolicies().get(0));
}
```

**Analyse und Begründung:**

Dieser Test ist gründlich, da er die Berechnung der Prämie für eine BASIC-Policy überprüft und sicherstellt, dass die Policy korrekt zum Kunden hinzugefügt wird. Es werden mehrere Assertions verwendet, um verschiedene Aspekte des Ergebnisses zu validieren.

#### 5.2.2.2 Positiv-Beispiel

**Code-Beispiel:**

```
@Test
void add_policy_with_customer_not_found() throws
CustomerNotFoundException, CustomerTooYoungException,
CarTooExpensiveException {
    when(customerRepository.findById(1)).thenReturn(Optional.empty());

    Policy policy = new Policy(1, PolicyStatus.ACTIVE,
PolicyProgram.BASIC, 10000);

    CustomerNotFoundException exception =
assertThrows(CustomerNotFoundException.class,() ->
policyManagement.addPolicyToCustomer(1, policy));
    assertEquals("The user with id 1 was not found.",
exception.getMessage());
}
```

**Analyse und Begründung:**

Dieser Test ist gründlich, da er nur überprüft, ob eine `CustomerNotFoundException` geworfen wird, wenn eine Policy für einen nicht existierenden Customer hinzugefügt werden soll.

### 5.2.3 ATRIP: Professional

#### 5.2.3.1 Positiv-Beispiel

**Code-Beispiel:**

```
@Test
void add_policy_with_too_high_car_value() {
    Customer customer = new TestCustomerDirector(new
Customer.Builder()).createMockUser();

when(customerRepository.findById(1)).thenReturn(Optional.of(customer))
;

    Policy policy = new Policy(1, PolicyStatus.ACTIVE,
PolicyProgram.BASIC, 120000);
    CarTooExpensiveException exception =
assertThrows(CarTooExpensiveException.class, () ->
policyManagement.addPolicyToCustomer(1, policy));

    assertEquals("Car value cannot be more than 100000!",
exception.getMessage());
}
```

**Analyse und Begründung:**

Dieser Test ist professionell, da er sicherstellt, dass die richtige Ausnahme geworfen wird, wenn der Autowert zu hoch ist. Er verwendet klare und verständliche Assertions und überprüft die Fehlermeldung der Ausnahme. Die Verwendung der Hilfsklasse `TestCustomerDirector` zur Erstellung von Customer-Objekten trägt zur Übersichtlichkeit bei, da sie es ermöglicht, auf einfache Weise unterschiedliche Customer-Objekte für diverse Testszenarien zu generieren.

#### 5.2.3.2 Negativ-Beispiel

**Code-Beispiel:**

```
@Test
void save_policy() {
    Customer customer = this.repository.findById(1).get();
    Policy policy = new Policy(0, PolicyStatus.ACTIVE,
PolicyProgram.DELUXE, 30000.0);

    customer.addPolicy(policy);
    Customer savedCustomer = repository.save(customer);

    assertNotNull(savedCustomer.getId());
    assertEquals(2,
this.repository.findById(savedCustomer.getId()).get().getPolicies().si
```

```
ze());
}
```

**Analyse und Begründung:**

Dieser Test ist nicht professionell, da das Policy Objekt manuell erstellt und somit nicht wiederverwendt werden kann. Dadurch müssen bei Änderungen der Policy Klasse sämtliche Tests angepasst werden.

### 5.3 Code Coverage

Die Code Coverage in diesem Projekt wird mit dem Tool **JaCoCo** gemessen. Eine hohe Code Coverage ist ein Indikator dafür, dass ein substantieller Teil des Codes durch automatisierte Tests abgedeckt ist, was potenziell die Fehlerwahrscheinlichkeit reduziert. Es ist jedoch wichtig zu betonen, dass eine hohe Testabdeckung allein keine Garantie für die Fehlerfreiheit darstellt. Fehlerhaft formulierte Assertions können dazu führen, dass bestehende Fehler nicht erkannt werden. Um die Robustheit des Codes zu gewährleisten, ist es unerlässlich, sowohl positive Tests (die das erwartete Verhalten verifizieren) als auch negative Tests (die die Fehlerbehandlung prüfen) zu implementieren.

**Analyse und Begründung:**
Der Schwerpunkt der Testaktivitäten lag auf der Applikations-Schicht, da diese die zentrale Geschäftslogik des Systems implementiert. Die Testsuite umfasst sowohl Unit-Tests als auch Integrationstests.
In der Domain-Schicht wurden primär die Value Objects durch Tests validiert. Die übrigen Klassen in dieser Schicht bestehen hauptsächlich aus Entitäten mit einfachen Getter- und Setter-Methoden, die kein zusätzliches Testen erfodern.
In der Adapter-Schicht wurde der Console-Adapter gezielt getestet, um dessen Funktionalität sicherzustellen.

**Domain-Schicht:**

“/assets/test_vo.png” could not be found.

**Applikations-Schicht:**

“/assets/test_application.png” could not be found.

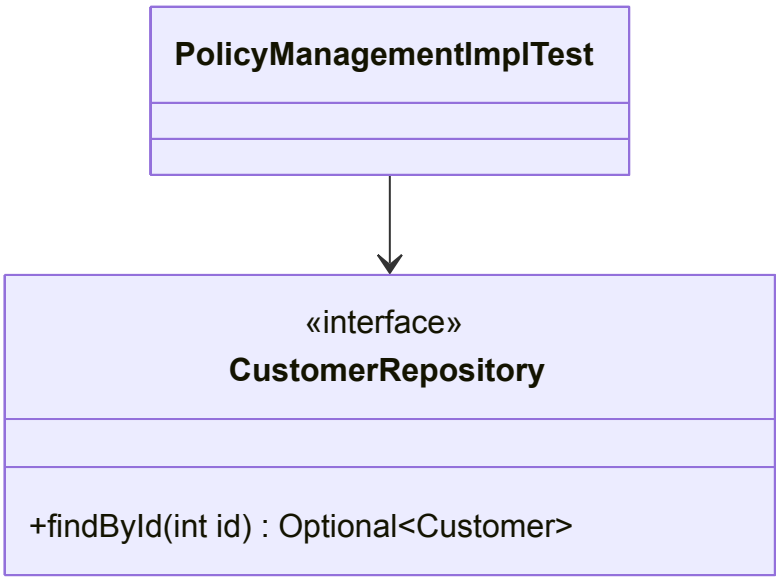
**Adapter-Schicht:**

“/assets/test_adapter.png” could not be found.

## 5.4 Fakes und Mocks

### 5.4.1 Mock-Objekt: `CustomerRepository`

In den Tests wird das `CustomerRepository` gemockt. Dies geschieht in mehreren Test Klassen, um die Geschäftslogik isoliert von der Implementierung des `CustomerRepository` zu trennen.




**Code-Beispiel:**

```
@ExtendWith(MockitoExtension.class)
class PolicyManagementImplTest {

  @Mock
  private CustomerRepository customerRepository;

  @InjectMocks
  private PolicyManagementImpl policyManagement;

  @Test
  void add_policy_basic_with_senior_driver() throws Exception {
    // Create a customer with age > 80
      Customer customer = new TestCustomerDirector(new
Customer.Builder()).createSeniorDriver();

when(customerRepository.findById(1)).thenReturn(Optional.of(customer))
;

      Policy policy = new Policy(1, PolicyStatus.ACTIVE,
PolicyProgram.BASIC, 10000);
```

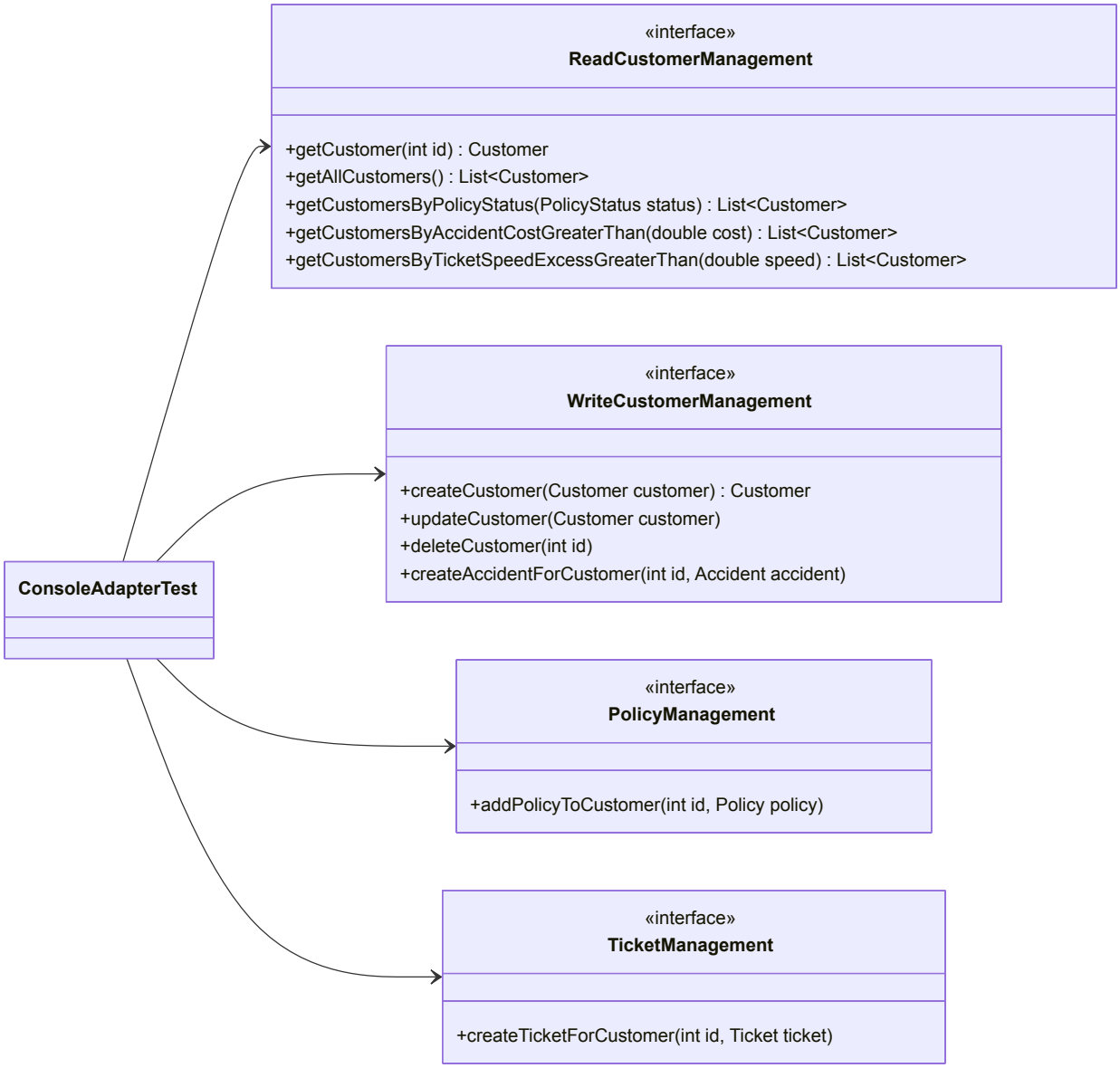
```
        policyManagement.addPolicyToCustomer(1, policy);

        assertEquals(550, policy.getPremium().getAmount());
        assertEquals(1, customer.getPolicies().size());
        assertEquals(policy, customer.getPolicies().get(0));
    }
}
```

**Beschreibung:**
Das Mock-Objekt *customerRepository* simuliert das Verhalten des `CustomerRepository`-Interfaces. Es wird verwendet, um die Abhängigkeit von der realen Implementierung zu isolieren und die Geschäftslogik von `PolicyManagementImpl` unabhängig zu testen. In diesem Test wird das Verhalten der `findById()`-Methode simuliert, sodass ein Senior-Customer zurückgegeben wird, ohne dass eine tatsächliche Datenbankabfrage durch den echten CustomerRepository stattfindet.

### 5.4.2 Mock-Objekt: `WriteCustomerManagement`

In den Tests des Adapters werden sämtliche Usecases gemockt. Dazu gehört das `WriteCustomerManagement`, aber auch `ReadCustomerManagement`, `PolicyManagement` und `TicketManagement`. Dies geschieht in der Testklasse `ConsoleAdapterTest`, um die Funktionsweise des Adapters unabhängig von der eigentlichen Geschäftlogik testen zu können. In den Tests des `ConsoleAdapter` werden außerdem die Methoden, die Nutzereingaben verlangen, gemockt, um automatisierte Tests zu ermöglichen, bei denen nicht manuell eine Eingabe erfolgen muss.




### Code-Beispiel:

```
@ExtendWith(MockitoExtension.class)
class ConsoleAdapterTest {

    @Mock
    private ReadCustomerManagement readCustomerManagement;

    @Mock
    private WriteCustomerManagement writeCustomerManagement;

    @Mock
    private PolicyManagement policyManagement;

    @Mock
    private TicketManagement ticketManagement;

    @InjectMocks
```

```
    private ConsoleAdapter consoleAdapter;

    @Test
        void createCustomer_success() {
            consoleAdapter = Mockito.spy(consoleAdapter);

            // Mocken der Eingabe-Methoden für automatisierte Tests

doReturn(1).doReturn(12).when(consoleAdapter).getIntInput(anyString())
;

doReturn("John").when(consoleAdapter).getStringInput(eq("Enter first
name: "));

doReturn("Doe").when(consoleAdapter).getStringInput(eq("Enter last
name: "));
            doReturn(LocalDate.of(2000, 1, 1)).when(consoleAdapter)
                .getDateInput(eq("Enter date of birth (YYYY-MM-DD):
"));

doReturn("john.doe@example.com").when(consoleAdapter).getStringInput(e
q("Enter email: "));

doReturn("Street").when(consoleAdapter).getStringInput(eq("Street:
"));

doReturn("City").when(consoleAdapter).getStringInput(eq("City: "));

doReturn("State").when(consoleAdapter).getStringInput(eq("State: "));

doReturn("12345").when(consoleAdapter).getStringInput(eq("Zip Code:
"));

doReturn("Country").when(consoleAdapter).getStringInput(eq("Country:
"));

            Customer createdCustomer = new CustomerDirector(new
Customer.Builder()).buildNew(1, "John", "Doe",
                    LocalDate.of(2000, 1, 1), "john.doe@example.com",
                    new Address("Street", "City", "State", "12345",
"Country"));

when(writeCustomerManagement.createCustomer(any(Customer.class))).then
Return(createdCustomer);

            consoleAdapter.start();

            String output = getOutput();
```

```
            assertTrue(output.contains("Customer created successfully
with ID: 1"));
            verify(writeCustomerManagement,
times(1)).createCustomer(any(Customer.class));
        }
}
```

**Beschreibung:**
Das Mock-Objekt *writeCustomerManagement* simuliert das Verhalten des `WriteCustomerManagement`-Interfaces. Es wird verwendet, um die Abhängigkeit von der realen Implementierung zu isolieren und die Adapter-Logik des `ConsoleAdapter` unabhängig von der Implementierung der Use-Cases zu testen. In den Tests wird das Verhalten der Methoden wie `createCustomer()`, `updateCustomer()`, etc. simuliert, ohne dass die tatsächliche Implementierung der Usecases (Business Logic) aufgerufen wird.

## 6. Domain-Driven-Design (DDD)

### 6.1 Ubiquitous Language

Bei den Unterabschnitten *6.1.1* und *6.1.2* handelt es sich um Vorarbeiten, die zu der **Ubiquitous Language** in *6.1.3* geführt haben.

#### 6.1.1 Entities

- Customer
  - natürliche Person, die Kunde bei der von der Anwendung verwalteten Autoversicherung ist
  - Vor- und Nachname, PersonId, Geburtsdatum (vielleicht als VO), Mail, Adresse
- Policy
  - eine Versicherung, die ein Kunde für ein Auto abgeschlossen hat
  - Policystatus, entweder aktiv oder nicht aktiv
  - Policyprogram: Verweis auf ein Programm
  - CarValue
  - Premium (vielleicht als VO)
- Policyprogram
  - welche Art von Policy abgeschlossen wird
  - wirkt sich auf den Preis der Policy aus
  - Id, Name, Beschreibung
- Policystatus
  - beschreibt den Status einer abgeschlossenen Versicherung

    - vorhandene Status: aktive, inaktive, abgelehnt
- Accidents
    - Unfälle, die ein Kunde begangen hat
    - Id, Schadenskosten, Datum
- Tickets
    - Verkehrsverstöße (in Bezug auf zu schnelles Fahren) von Kunden
    - Id, Datum, Geschwindigkeitsüberschreitung

### 6.1.2 Nutzer

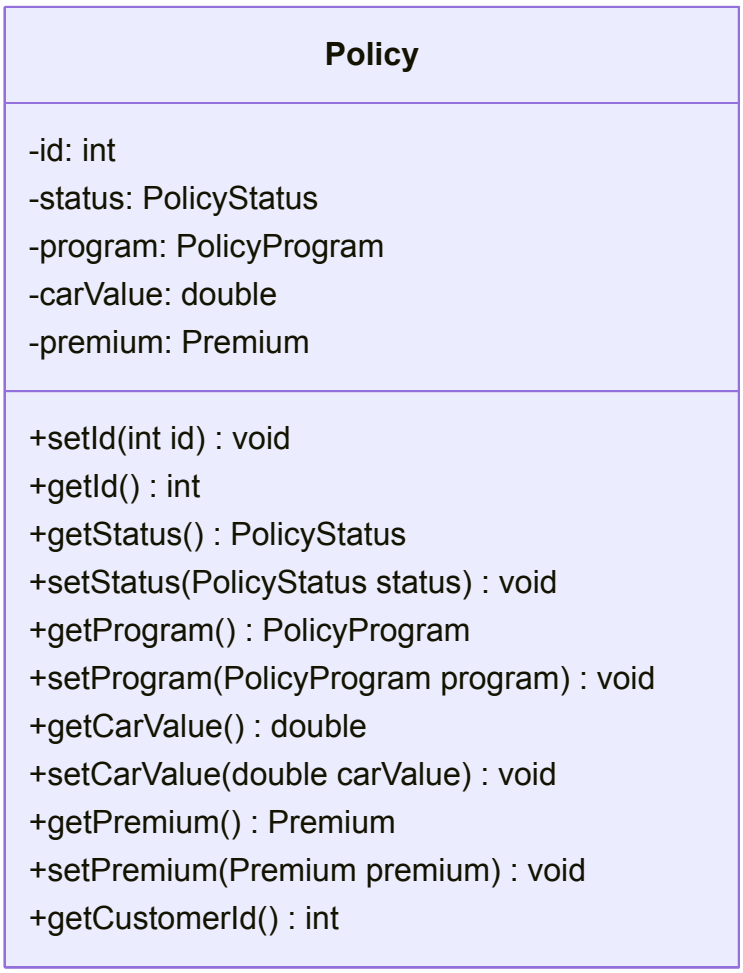
- Mitarbeiter der Versicherungsfirma
    - Kundenverwaltung
        - neuen Kunden im System anlegen
        - Policy für einen Kunden erstellen
        - alte Kunden und ihre Policys anzeigen
    - Accident hinzufügen
        - bei Kundenmeldung über einen Accident soll dieser in das System eingetragen werden
        - Änderungen an der spezifischen Policy des Kunden soll ersichtlicht werden
            - bis zu einer bestimmten Menge an Unfällen wird es teurer, die Policies können aber auch gekündigt werden
    - Ticket hinzufügen
        - bei Kundenmeldung über ein Ticket soll dieses in das System eingetragen werden
        - Änderungen an allen Policys des Kunden sollen ersichtlich werden
            - bis zu einer bestimmten Menge an Tickets oder ab einer bestimmten Geschwindigkeitsüberschreitung wird es teurer, die Policies können aber auch gekündigt werden

### 6.1.3 Tabelle

| Bezeichnung | Bedeutung | Begründung |
|---|---|---|
| Policy | eine Versicherung, die ein Kunde für ein Auto abgeschlossen hat | Ein Kunde schließt einen Vertrag für jede seiner Versicherungen ab (Insurance Policy). Bei einer Autoversicherung werden nur diese Art von Verträgen verwaltet, deswegen die kürzere Bezeichnung. |
| Ticket | Verkehrsverstöße von Kunden für zu schnelles Fahren | Bei einer einer Autoversicherung sind für die Kostenberechnung einer Policy Geschwindigkeitsüberschreitungen releveant, da dadurch das Risiko eines Schadens erhöht wird. |

| Bezeichnung | Bedeutung | Begründung |
|---|---|---|
| | | Andere Arten von Verkehrsvergehen, wie falsches Parken, werden dabei nicht berücksichtigt. |
| Customer | natürliche Person, die Kunde bei der von der Anwendung verwalteten Autoversicherung ist | Kunden schließen einen Vertrag bei der Autoversicherung ab. Die Nutzer der Anwendung sind Mitarbeiter der Autoversicherung und tragen die Daten für die jeweiligen Kunden in das System ein. |
| Premium | die monatlichen Kosten einer Policy, um diese aktiv zu halten | Im Rahmen einer Versicherung wird Premium als der monatlich zu entrichtende Betrag definiert |

### 6.2 Entities - Policy Entity

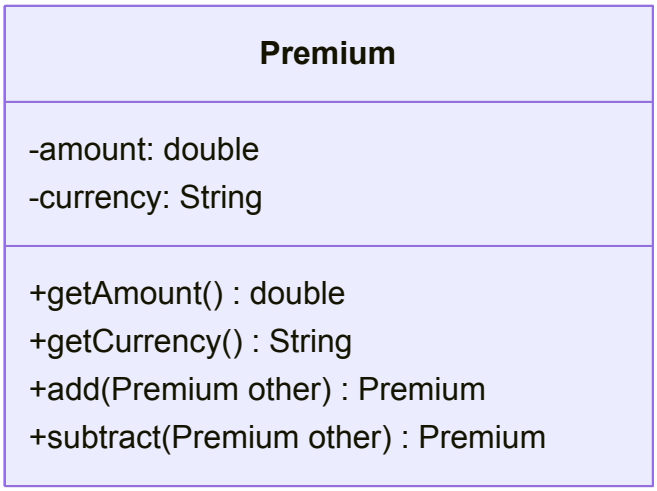



**Beschreibung**
Die Entität *Policy* repräsentiert eine VersicherungsPolicy, die ein Kunde für ein spezifisches Auto abgeschlossen hat. Sie enthält wesentliche Informationen wie den Status der Policy, das gewählte Versicherungsprogramm, den Wert des versicherten Autos und die monatlichen Kosten.

**Begründung des Einsatzes:**
Policy wird als Entity modelliert, weil:

1. Sie eine eindeutige Identität hat (durch die id).
2. Sie einen Lebenszyklus hat (der Status kann verändert werden).
3. Sie sich im Laufe der Zeit ändern kann (z. B. Änderung der Kosten), behält aber ihre Identität.
4. Sie eine zentrale Rolle im Modell spielt und mit anderen Entitäten (wie Customer) in Beziehung steht.

### 6.3 Value Objects - Premium Value Object

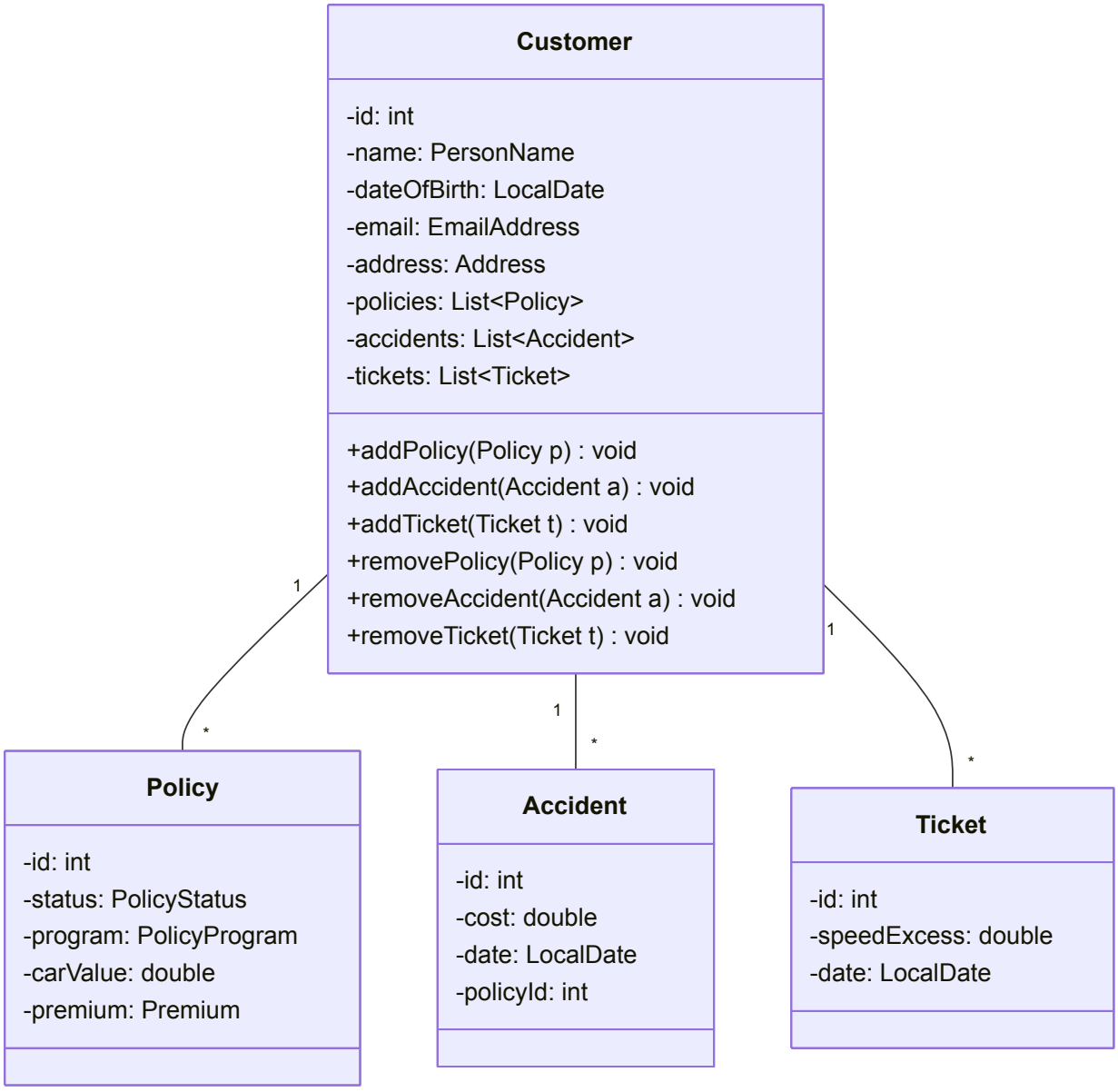



**Beschreibung:**
Das Value Object *Premium* repräsentiert den Geldbetrag, den ein Kunde monatlich für seine VersicherungsPolicy zahlt. Es kapselt den Betrag und die Währung.

**Begründung des Einsatzes:**
Premium wird als Value Object modelliert, weil:

1. Es keine eigene Identität hat - zwei Premium-Objekte mit dem gleichen Betrag und der gleichen Währung sind austauschbar.
2. Es unveränderlich (immutable) ist - Änderungen erzeugen ein neues Objekt.
3. Es Verhaltenslogik enthält (z. B. Addition, Subtraktion), die für alle Premium-Objekte gleich ist.
4. Es ein konzeptionell zusammengehöriges Paar von Werten (Betrag und Währung) repräsentiert.

### 6.4 Aggregates - Customer Aggregate

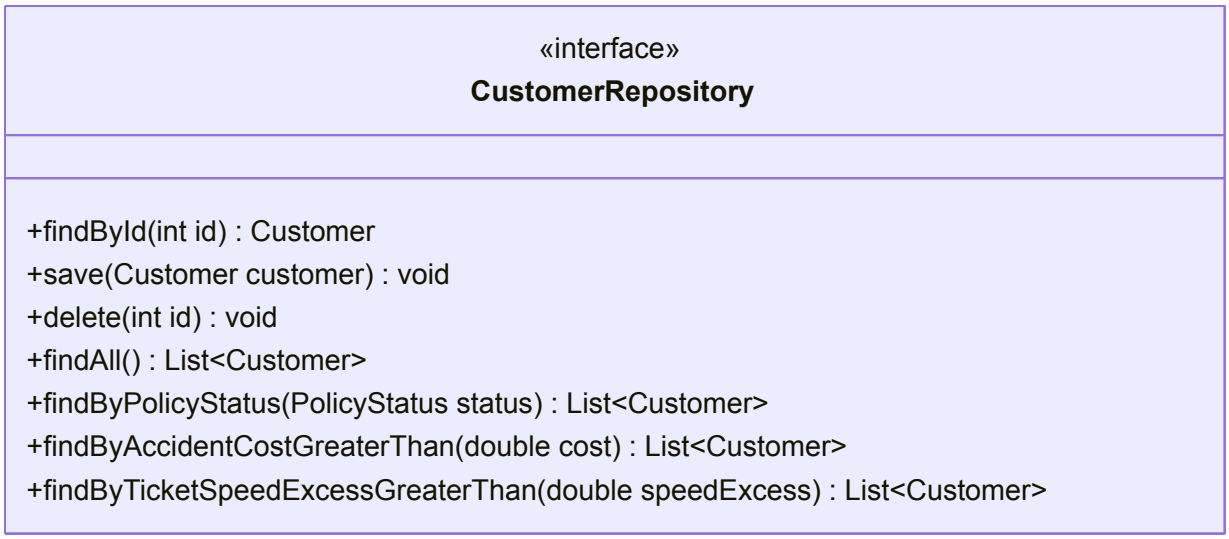



**Beschreibung:**
Das Aggregate *Customer* gruppiert die Entitäten Customer, Policy, Accident und Ticket. Customer fungiert als Aggregate Root und hat direkte Verbindungen zu seinen Policies, Accidents und Tickets.

**Begründung des Einsatzes:**
Ein Aggregat wird hier eingesetzt, weil:

1. Es eine logische Gruppierung zusammengehöriger Entitäten darstellt. Ein Kunde hat Policys, Unfälle und Tickets, die alle direkten Einfluss auf die Versicherungskosten haben.
2. Es die Konsistenz der Daten sicherstellt. Änderungen an Policies, Accidents oder Tickets müssen immer im Kontext des zugehörigen Kunden erfolgen.
3. Es die Komplexität reduziert, indem es einen einzelnen Zugriffspunkt (Customer als Aggregate Root) für zusammengehörige Daten bietet.

### 6.5 Repositories - Customer Repository




**Beschreibung:**
Das Repository *Customer* ist verantwortlich für die Persistenz und das Abrufen des Aggregates Customer. Es bietet Methoden zum Finden, Speichern, Aktualisieren und Löschen von Kunden.

**Begründung des Einsatzes:**
Ein Repository für Customer wird eingesetzt, weil:

1. Es die Datenzugriffslogik von der Geschäftslogik trennt.
2. Es eine Abstraktion der Datenpersistenz bietet, wodurch die zugrunde liegende Datenbank ohne Änderungen an der Geschäftslogik geändert werden kann.
3. Es ermöglicht, komplexe Abfragen zu kapseln (z. B. findByName).
4. Es unterstützt das Prinzip der Aggregate Root in DDD, da Customer ein Aggregate Root ist.

## 7. Refactoring

### 7.1 Code Smells

#### 7.1.1 Long Method

**Code-Beispiel:**
Die Methode `addPolicyToCustomer` in `PolicyManagementImpl` ist zu lang und enthält zu viele Verantwortlichkeiten.

```
public void addPolicyToCustomer(int customerId, Policy policy)
        throws CustomerNotFoundException, CustomerTooYoungException,
```

```
CarTooExpensiveException {
    Customer customer = customerRepository.findById(customerId)
            .orElseThrow(() -> new
CustomerNotFoundException(customerId));

    if (customer.getAge() < 18) {
        throw new CustomerTooYoungException(customerId);
    }

    if (policy.getCarValue() > 100000) {
        throw new CarTooExpensiveException(policy.getCarValue());
    }

    customer.addPolicy(policy);
    customerRepository.save(customer);
}
```

**Möglicher Lösungsweg:**
Aufteilung der Methode in kleinere Methoden, die jeweils eine einzelne Verantwortung haben.

```
public void addPolicyToCustomer(int customerId, Policy policy)
        throws CustomerNotFoundException, CustomerTooYoungException,
CarTooExpensiveException {
    Customer customer = findCustomerById(customerId);
    validateCustomerAge(customer);
    validateCarValue(policy);
    addPolicyAndSaveCustomer(customer, policy);
}

private Customer findCustomerById(int customerId) throws
CustomerNotFoundException {
    return customerRepository.findById(customerId)
            .orElseThrow(() -> new
CustomerNotFoundException(customerId));
}

private void validateCustomerAge(Customer customer) throws
CustomerTooYoungException {
    if (customer.getAge() < 18) {
        throw new CustomerTooYoungException(customer.getId());
    }
}

private void validateCarValue(Policy policy) throws
CarTooExpensiveException {
    if (policy.getCarValue() > 100000) {
```

```
            throw new CarTooExpensiveException(policy.getCarValue());
        }
    }

    private void addPolicyAndSaveCustomer(Customer customer, Policy
    policy) {
        customer.addPolicy(policy);
        customerRepository.save(customer);
    }
```

### 7.1.2 Duplicated Code

**Code-Beispiel:**
In der `PolicyManagementImpl` und `TicketManagementImpl` gibt es ähnliche Logik für das Abrufen und Überprüfen von Kunden.

```
class PolicyManagementImpl {
    // ...
    private Customer getCustomer(int customerId) throws
CustomerNotFoundException {
        return customerRepository.findById(customerId).orElseThrow(()
-> new CustomerNotFoundException(customerId));
    }
}
```

```
class TicketManagementImpl {
    // ...
    private Customer getCustomer(int customerId) throws
CustomerNotFoundException {
        return customerRepository.findById(customerId).orElseThrow(()
-> new CustomerNotFoundException(customerId));
    }
}
```

**Möglicher Lösungsweg:**
Extrahieren der gemeinsamen Logik in eine Hilfsklasse.

```
package de.sri.application.usecases;

public class CustomerHelper {

    public static Customer getCustomer(CustomerRepository
customerRepository, int customerId) throws CustomerNotFoundException {
        return customerRepository.findById(customerId).orElseThrow(()
```

```
-> new CustomerNotFoundException(customerId));
    }

}
```

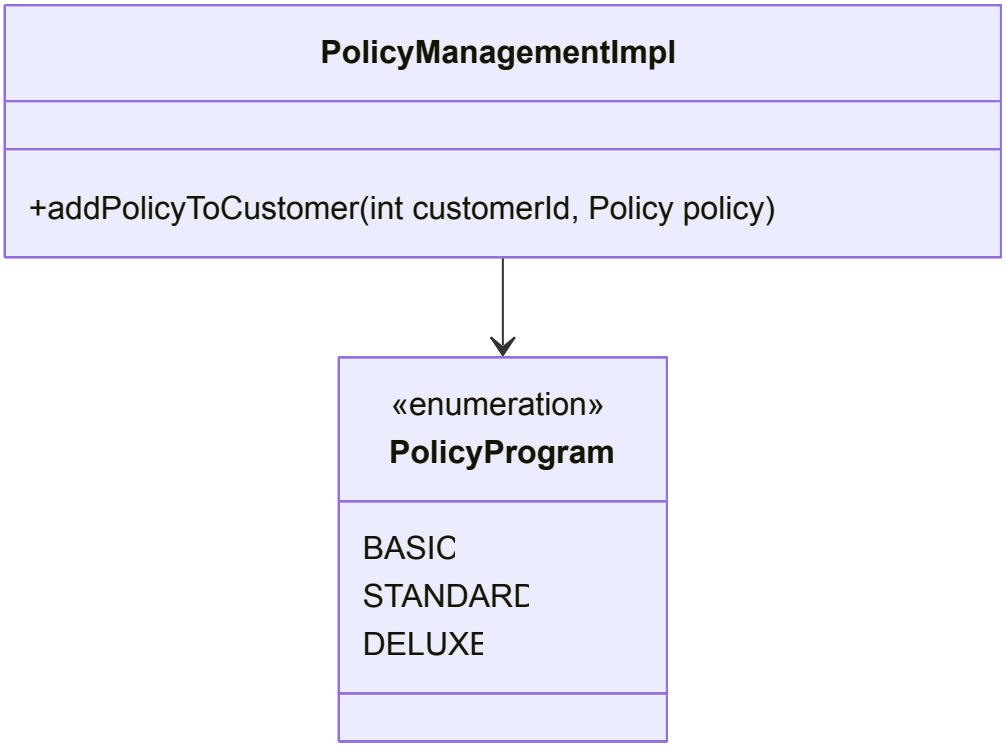
## 7.2 Refactorings

### 7.2.1 Replace Conditional with Polymorphism

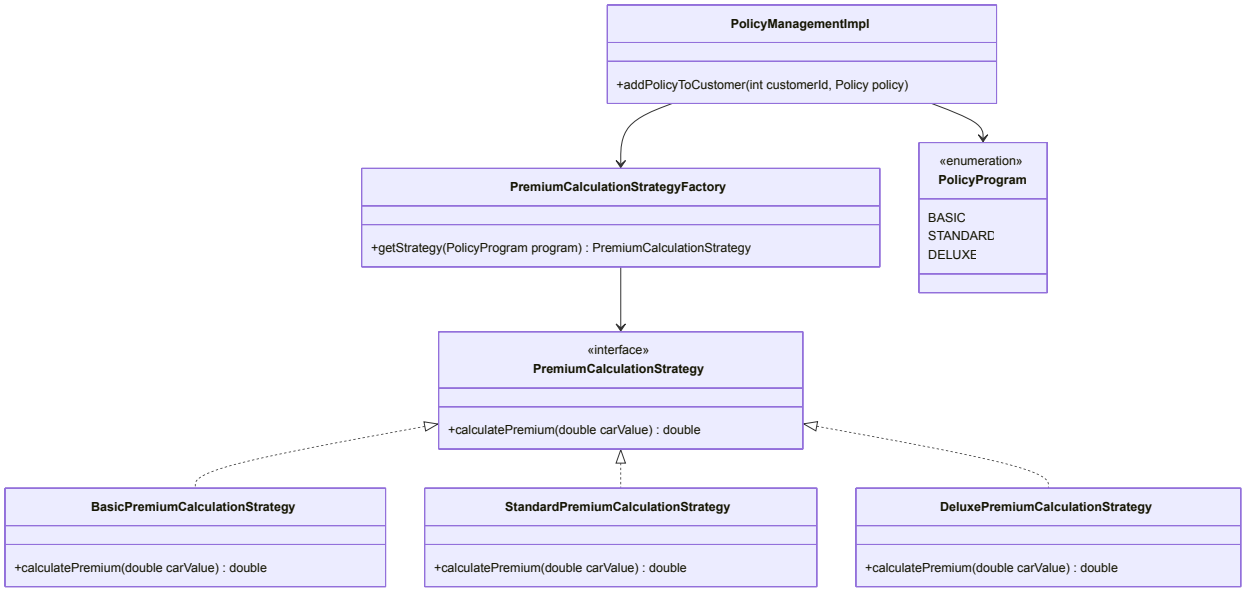
**Begründung:**
Das Refactoring "Replace Conditional with Polymorphism" wird angewendet, um die Wartbarkeit und Erweiterbarkeit des Codes zu verbessern. Anstatt eine lange `switch`-Anweisung zu verwenden, um die Premium-Berechnung basierend auf dem `PolicyProgram` zu bestimmen, wird das Strategy-Pattern verwendet. Dies ermöglicht es, neue Berechnungsstrategien hinzuzufügen, ohne den bestehenden Code zu ändern.

Commit (ursprünglich eingeführt): `d8729c9c1f914fe12341021ee10d92481abb4f7b`
Commit (Switch der Factory mit HashMap ersetzt): `18a713665a859cbaaeead7b04785788d18e5752d`

**UML Vorher:**




**UML Nachher:**

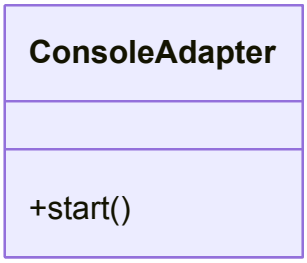



### 7.2.2 Extract Method

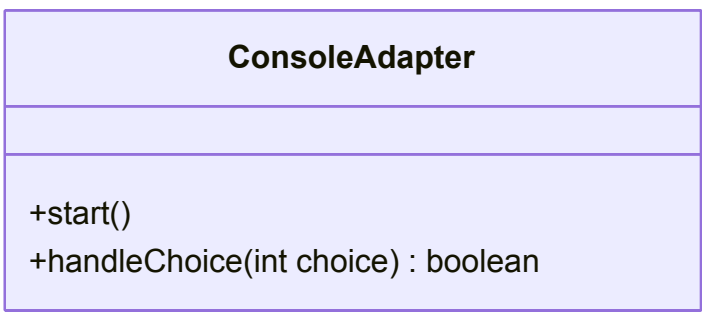
**Begründung:**
Das Refactoring "Extract Method" wurde angewendet, um die Lesbarkeit und Wartbarkeit des Codes zu verbessern. Durch das Extrahieren der Menüauswahl-Logik in eine separate Methode `handleChoice` wird die `start`-Methode vereinfacht und die Verantwortlichkeiten klarer getrennt.

Commit: 9a478dff365058233dde2b14bad25be8f8b1495c

**UML Vorher:**




**UML Nachher:**



**Code Vorher:**

```
public void start() {
    boolean running = true;
    while (running) {
        try {
            printMainMenu();
            int choice = getIntInput("Choose an option: ");

            switch (choice) {
                case 1:
                    createCustomer();
                    break;
                // ...
                case 12:
                    running = false;
                    break;
                default:
                    System.out.println("Invalid option. Please try
again.");
            }
        } catch (BaseDomainException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) {
            // In einem realen Beispiel sollte man hier loggen.
            System.out.println("An unexpected error occurred. Try
again or contact support.");
        }
    }
}
```

**Code Nachher:**

```
public void start() {
    boolean running = true;
    while (running) {
        try {
            printMainMenu();
            int choice = getIntInput("Choose an option: ");
            running = handleChoice(choice);
        } catch (BaseDomainException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) {
            // In einem realen Beispiel sollte man hier loggen.
            System.out.println("An unexpected error occurred. Try
again or contact support.");
```

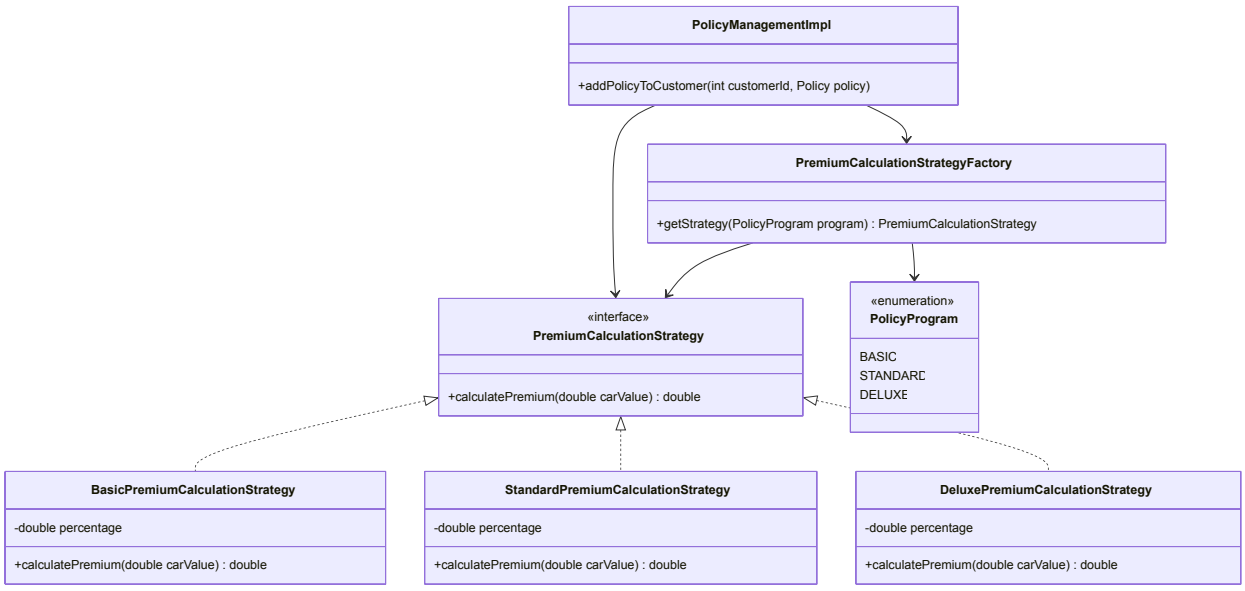
```
            }
        }
    }

    private boolean handleChoice(int choice) throws BaseDomainException {
        switch (choice) {
            case 1:
                createCustomer();
                break;
            // ...
            case 12:
                return false;
            default:
                System.out.println("Invalid option. Please try again.");
        }
        return true;
    }
```

## 8. Design Patterns

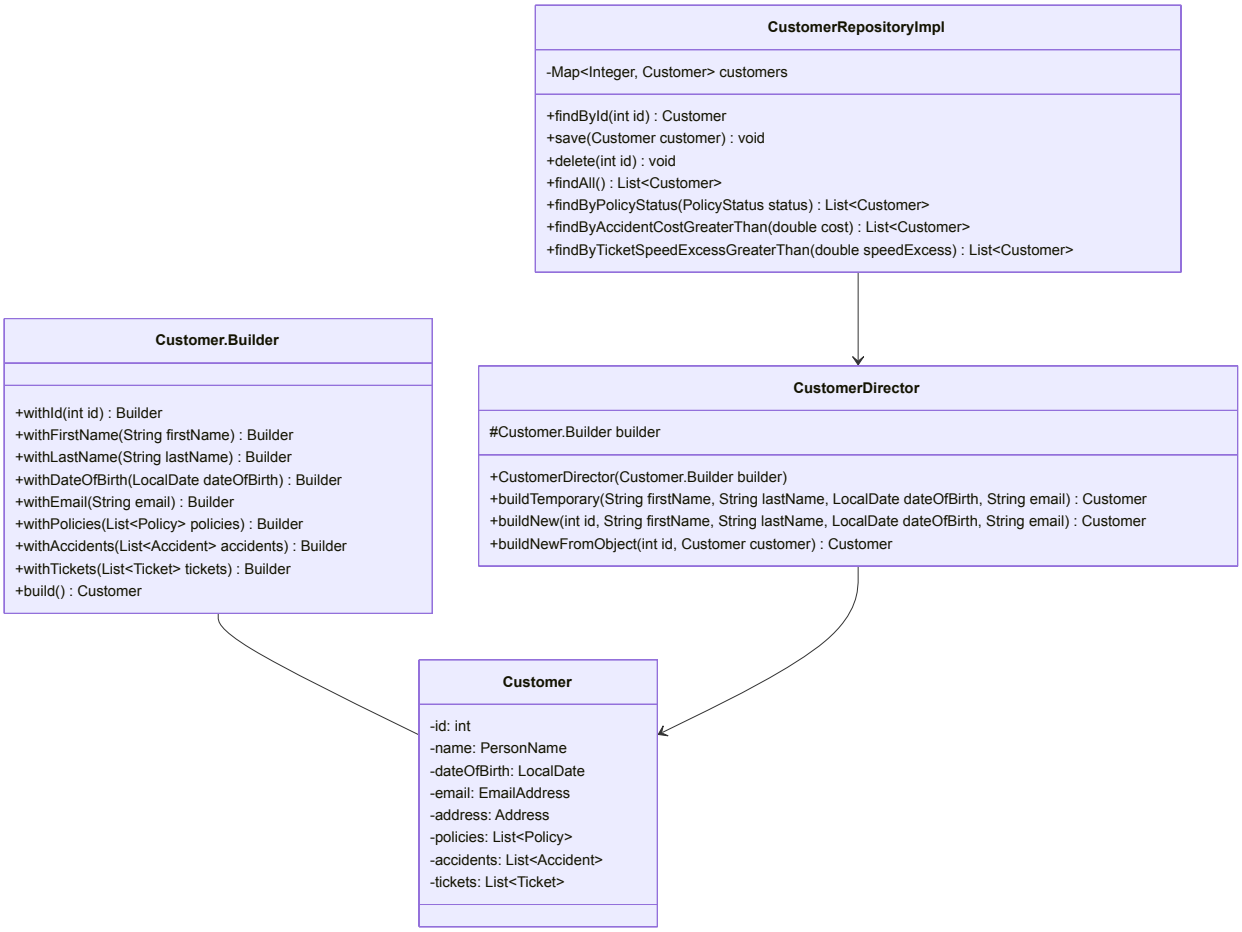
### 8.1 Strategy Pattern

**Begründung:**
Das Strategy Pattern wird verwendet, um verschiedene Berechnungsstrategien für Premiums zu kapseln. Dies ermöglicht es, die Berechnungslogik für verschiedene Policy-Programme (BASIC, STANDARD, DELUXE) zu variieren, ohne den Code der PolicyManagementImpl-Klasse zu ändern.




### 8.2 Builder Pattern

**Begründung:**
Das Builder Pattern wird verwendet, um die Erstellung komplexer Customer-Objekte zu vereinfachen. Der CustomerDirector nutzt den Builder, um verschiedene Arten von Customer-Objekten zu erstellen, was die Lesbarkeit und Wartbarkeit des Codes verbessert.




---

Größten Respekt, falls du hier angekommen bist ;)