




# Hotelbuchungssystem

Kai Schablowsky & Manuel Franz

# Inhalte der Präsentation

- Überblick & Einführung
- Softwarearchitektur
- Solid
- Weitere Prinzipien
- Unit Tests
- Domain Driven Design
- Refactoring
- Entwurfsmuster



# Einführung

# Einführung - Definition

- Definition unserer Applikation  
Hotelbuchungssystem für ein Hotel für die lokale Verwaltung von Hotelprozessen
- Unsere Features
  - Anlegen von Gästen
  - Buchung von Zimmern
  - Buchung von Restauranttischen
  - Buchung von Parkplätzen
  - Buchung von Aktivitäten
  - Analysen durchführen

# Einführung - Demo

- Demo unseres Hotelbuchssystem

```
C:\Users\manue\Documents\DHBW\Semester_5_30.09.2024_23.12.2024\advanced_SWE\Hotelbuchungssystem\presentation>java -jar Hotelbuchungssystem.jar createUser --firstName Kai --lastName Schablowsky --mail test@mail.com -- country Deutschland --postalCode 76149 --city Karlsruhe --street Hauptstrasse -- houseNumber 1
>--Hotelbuchungssystem--<
User created!
```

```
C:\Users\manue\Documents\DHBW\Semester_5_30.09.2024_23.12.2024\advanced_SWE>cloc-2.02.exe .\Hotelbuchungssystem\src\main\
61 text files.
61 unique files.
1 file ignored.
```

```
github.com/AlDanial/cloc v 2.02 T=0.40 s (152.4 files/s, 9922.8 lines/s)
```

Language	files	blank	comment	code
Java	60	618	83	3239
XML	1	1	1	29
SUM:	61	619	84	3268

# Einführung - Techstack

- Technologien
  - Java 21
    - Bibliotheken:
      - SQLite JDBC
      - Hibernate ORM Hibernate Community Dialects
      - SLF4J Simple Provider
    - Maven
    - SQLite
  - IDE (Integrated development environment)
    - IntelliJ IDEA



# Softwarearchitektur

# Softwarearchitektur (1)

- MAIN – Aufruf des Programms inklusive der Datenbankinitialisierung
- CLI – Gesamte Logik und Funktionalität der CLI
  - enums – Auflistung der verschiedenen Haupt- und Subbefehle
  - interaction\_classes – Klassen für die verschiedenen Funktionen aufgeteilt
- Service – Implementierung der Geschäftslogik
  - Buchungen
  - Analysen & Rankings
- Repository – Repositories für verschiedene Interaktionen
- Model - Alle verwendeten Entitäten



**„Schichtenmodell“ als angewandte  
Softwarearchitektur**



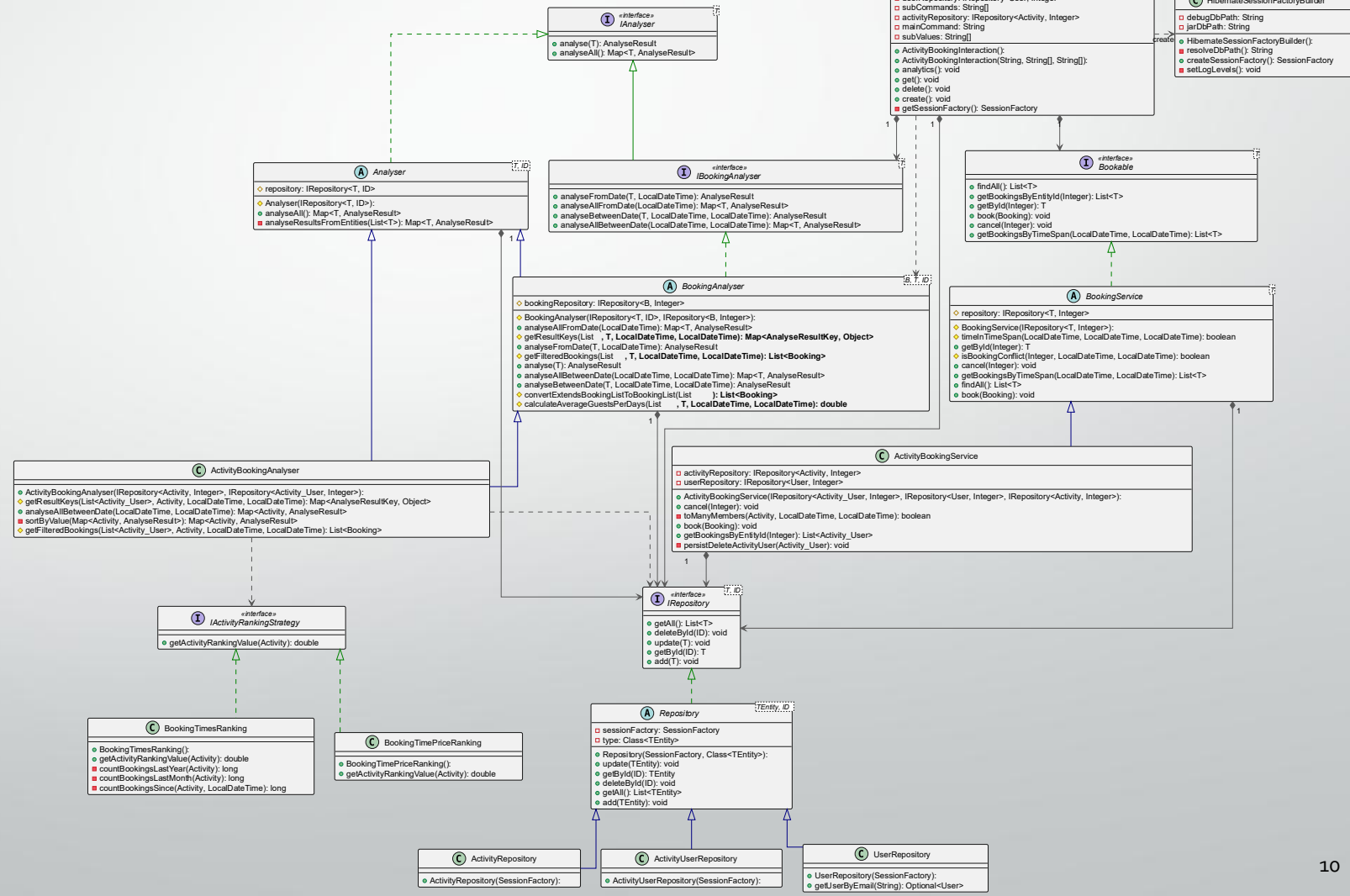
# Softwarearchitektur (2)

Wieso wir ein einfaches Schichtenmodell verwenden:

- Jede Schicht übernimmt eine klar definierte Aufgabe
- Schichten hängen nur von unteren Schichten ab

=> bessere Strukturierung, Wartbarkeit und Wiederverwendbarkeit

# Gesamtes UML Überblick – ActivityBooking



# Domaincode

Beinhaltet die Geschäftslogik (Buchungen & Analysen)

## **Beispiel Domain Code:**

- Buchung eines neuen Hotelzimmers
- Buchung eines neuen Restauranttisch
- Buchung einer neuen Aktivität
- Analyse von Daten z.B. Auslastung eines Zimmers

# Dependency Rule

Positiv Beispiel Dependency Rule:

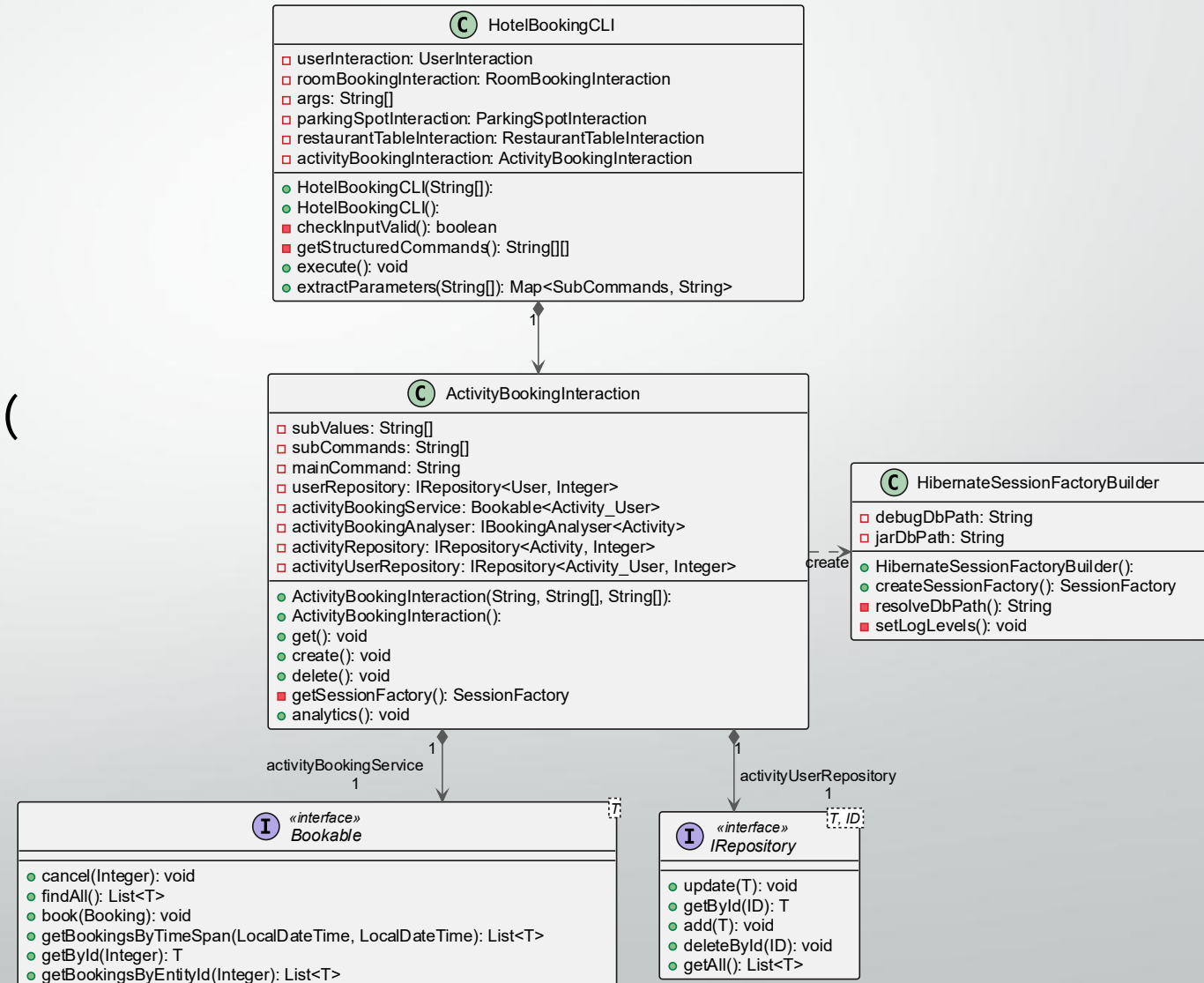
- ActivityBookingService
  - Klassen der Schicht Service hängen ausschließlich von untereren Schichten (Repositories) ab
  - Tiefe Schichten hängen nicht von oberen Schichten ab
  - Klar getrennte Abhängigkeiten
  - Auf den Service wird nur von einer noch weiter oben liegenden Schicht zugegriffen (CLI)



# Dependency Rule

## Negativ Beispiel ActivityBookingInteraction

Verletzung durch den Aufruf des  
HibernateSessionFactoryBuilder (in der oberen Schicht  
MAIN/Initialisierung)





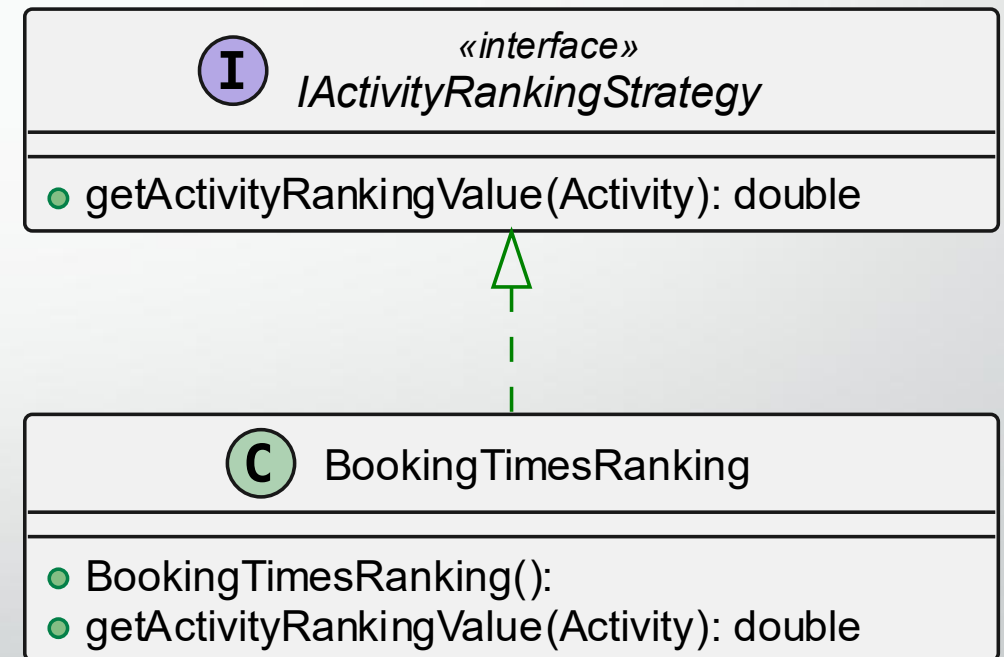
SOLID

# SRP – Single Responsibility Princip

Positivbeispiel – BookingTimesRanking der Aktivitäten

Aufgabe:

- Ermitteln des Ranking-Werts einer Aktivität





# SRP – Single Responsibility Princip

Negativbeispiel –  
HibernateSessionFactoryBuilder

Aufgaben:

- Erstellung einer Session Factory
- Einstellung des Loglevels
- Erstellung der Hibernate-Konfiguration
- DB-Pfad bestimmen



HibernateSessionFactoryBuilder

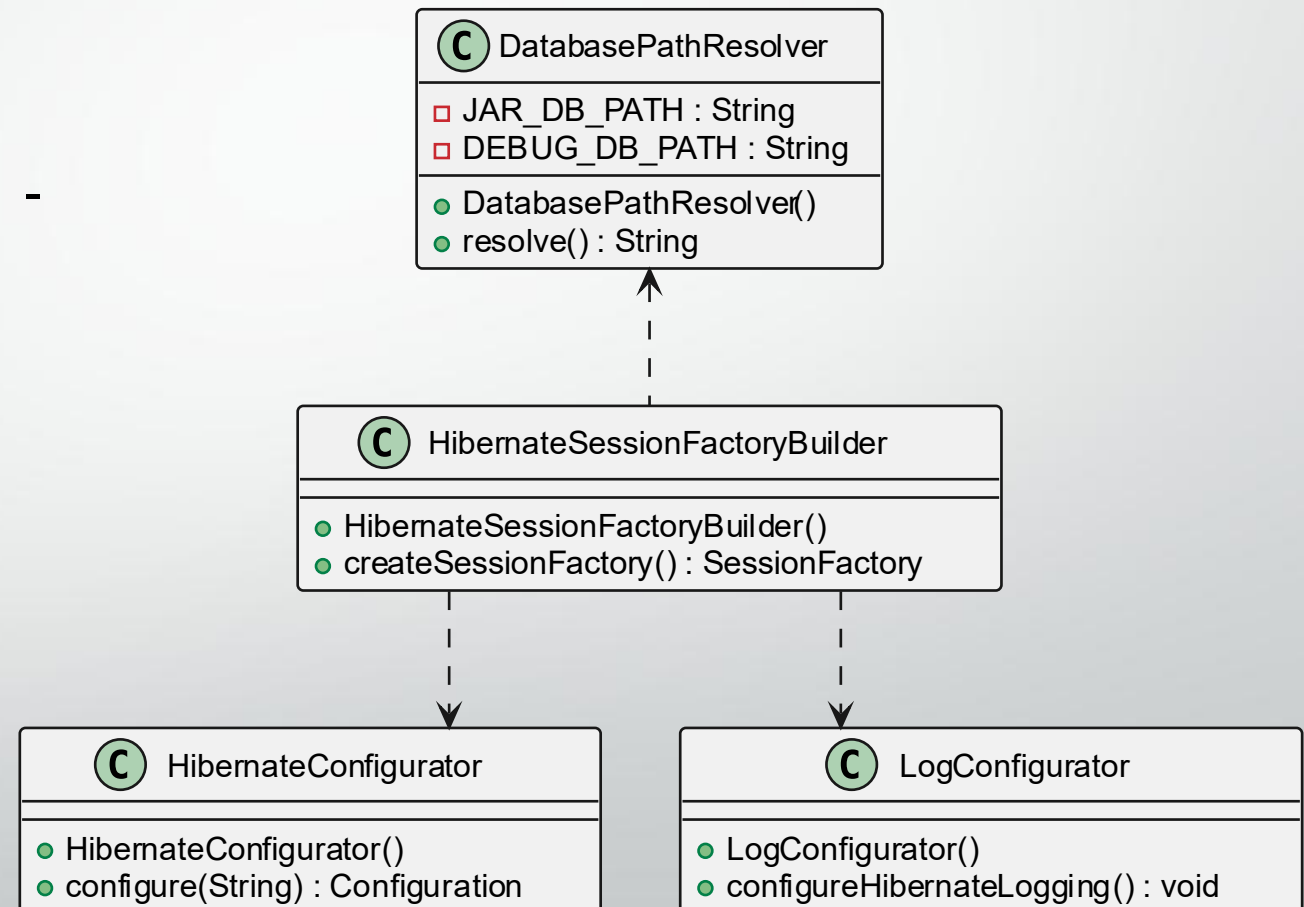
- HibernateSessionFactoryBuilder():
- resolveDbPath(): String
- createSessionFactory(): SessionFactory
- setLogLevels(): void

Siehe Code

# SRP – Single Responsibility Princip

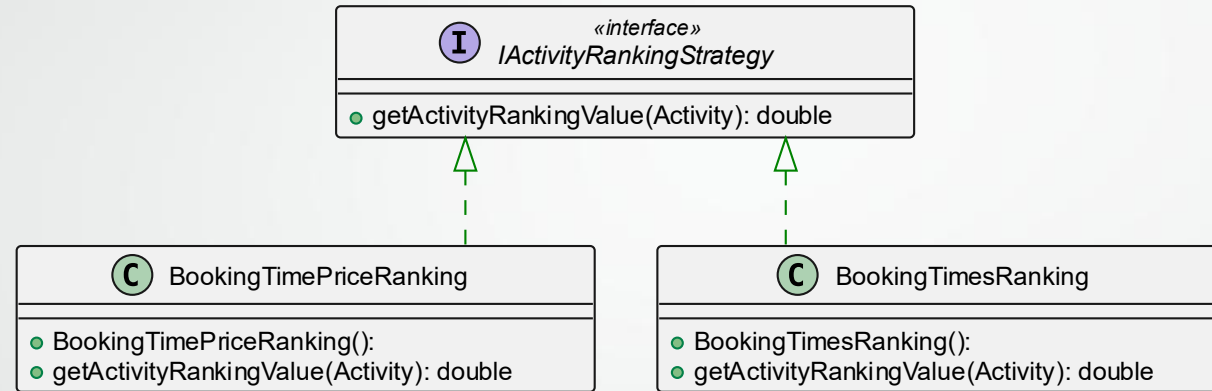
Negativbeispiel –  
HibernateSessionFactoryBuilder -  
Lösung

Jede Teilaufgabe wurde in eine  
eigene Klasse ausgelagert



# OCP – Open/Closed Principle

## Positivbeispiel – Rankingstruktur der Aktivitäten



Warum erfüllt:

- Interface definiert Verhalten → neue Ranking-Strategien können hinzugefügt werden, ohne bestehende Klassen zu ändern.
- Klassen (`BookingTimePriceRanking`, `BookingTimesRanking`) implementieren das Interface → Erweiterung durch neue Klassen möglich.
- Polymorphie wird genutzt → Aufrufer (z. B. ein Ranking-Service) muss nur das Interface kennen, nicht die konkreten Implementierungen. Keine Änderungen der Codebasis nötig

# OCP – Open/Closed Principle

Warum ist OCP hier sinnvoll?

- Flexible Erweiterbarkeit: Neue Bewertungslogiken (z. B. "Kundenbewertungen", "Beliebtheit") können einfach durch neue Klassen ergänzt werden.
- Keine Gefahr bestehende Logik zu zerstören: Bestehende Ranking-Strategien bleiben unberührt.
- Einheitlicher Zugriff: Alle Strategien haben die gleiche Methode (`getActivityRankingValue`), was den Code einfach und konsistent hält.
- Leichte Testbarkeit: Einzelne Strategien können separat getestet werden.

# OCP – Open/Closed Principle

## Negativbeispiel – HotelBookingCLI

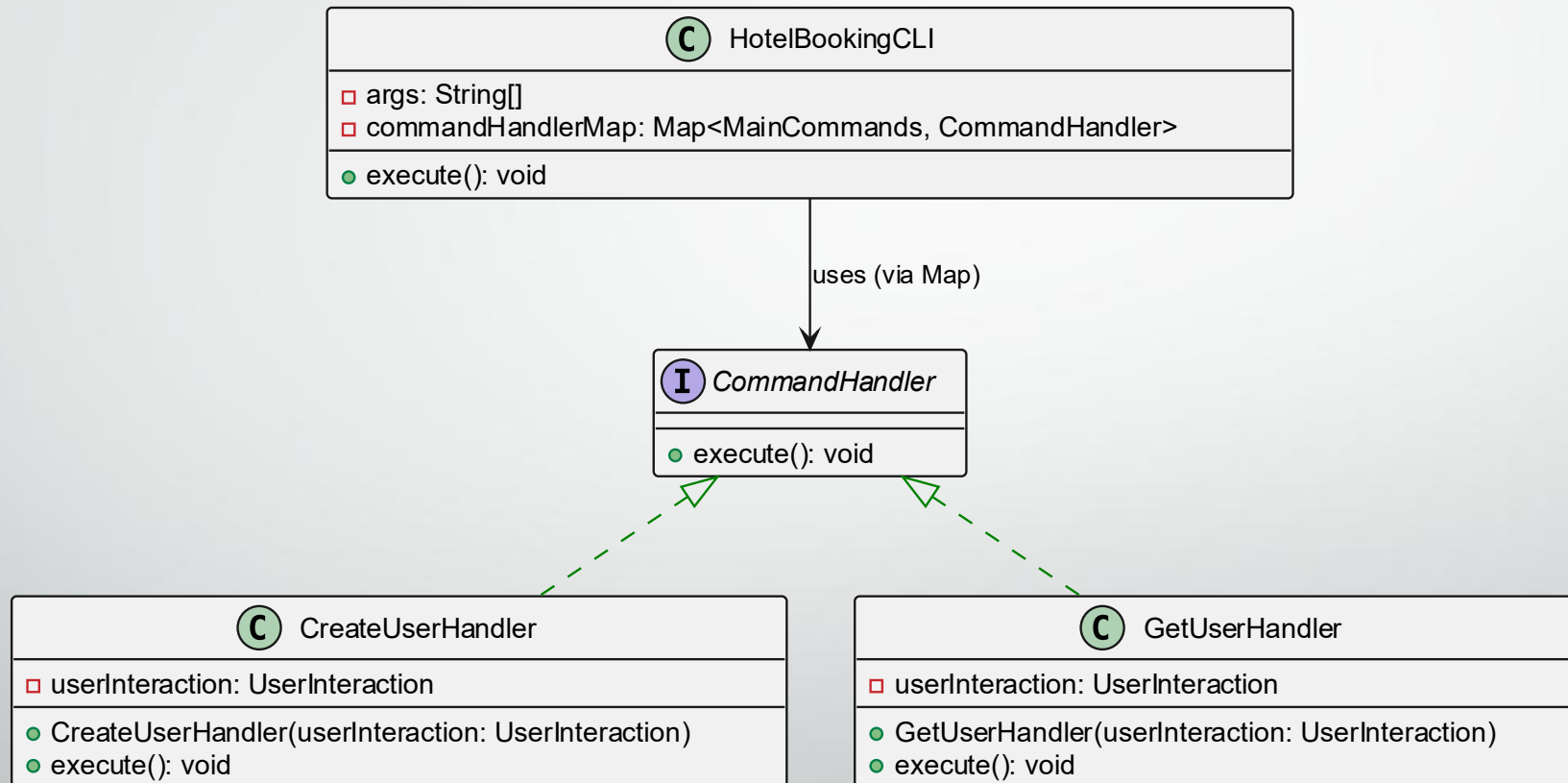
Siehe Code  
(switch-case)

C HotelBookingCLI
structuredCommands: String[][]
<ul style="list-style-type: none"><li>● HotelBookingCLI():</li><li>● HotelBookingCLI(String[]):</li><li>● execute(): void</li><li>● extractParameters(String[]): Map&lt;SubCommands, String&gt;</li><li>■ checkInputValid(): boolean</li></ul>

Warum nicht erfüllt:

- Großer switch-case Block:  
Jedes Mal, wenn ein neuer MainCommand (z. B. UPDATE\_USER) hinzugefügt wird, muss HotelBookingCLI geändert werden.  
(z.B. neue case-Zeilen schreiben.)
- Direkte Kenntnis aller Interaction-Klassen (UserInteraction, RoomBookingInteraction, usw.):  
HotelBookingCLI ist abhängig von allen spezifischen Implementierungen → Erweiterung bedeutet Änderung dieser Klasse.

# OCP – Open/Closed Principle



# OCP – Open/Closed Principle

## HotelBookingCLI – execute()

```
Map<MainCommands, CommandHandler> commandHandlerMap = new HashMap<>();

UserInteraction userInteraction = new UserInteraction(mainCommand, subValues, totalSubCommands);
RoomBookingInteraction roomBookingInteraction = new RoomBookingInteraction(mainCommand, subValues, totalSubCommands);
ParkingSpotInteraction parkingSpotInteraction = new ParkingSpotInteraction(mainCommand, subValues, totalSubCommands);
RestaurantTableInteraction restaurantTableInteraction = new RestaurantTableInteraction(mainCommand, subValues, totalSubCommands);
ActivityBookingInteraction activityBookingInteraction = new ActivityBookingInteraction(mainCommand, subValues, totalSubCommands);

commandHandlerMap.put(MainCommands.CREATE_USER, new CreateUserHandler(userInteraction));
commandHandlerMap.put(MainCommands.GET_USER, new GetUserHandler(userInteraction));

CommandHandler handler = commandHandlerMap.get(matchingCommand);
if (handler != null) {
    handler.execute();
} else {
    System.out.println("Invalid command!");
}
```

```
public interface CommandHandler {
    void execute();
}
```

```
public class CreateUserHandler implements CommandHandler {
    private final UserInteraction userInteraction;

    public CreateUserHandler(UserInteraction userInteraction) {
        this.userInteraction = userInteraction;
    }

    @Override
    public void execute() {
        userInteraction.create();
    }
}
```

# OCP – Open/Closed Principle

Negativbeispiel – HotelBookingCLI - Lösung

Verbesserung:

- neues gemeinsames Interface, z.B. CommandHandler.
- Jede Aktion (z.B. CreateUserHandler, GetUserHandler, etc.) implementiert dieses Interface.
- HotelBookingCLI hält nur eine Map<MainCommands, CommandHandler>.
- Bei neuen Commands → nur neue Handlerklasse + Registrierung in Map, keine Änderung am CLI-Code!

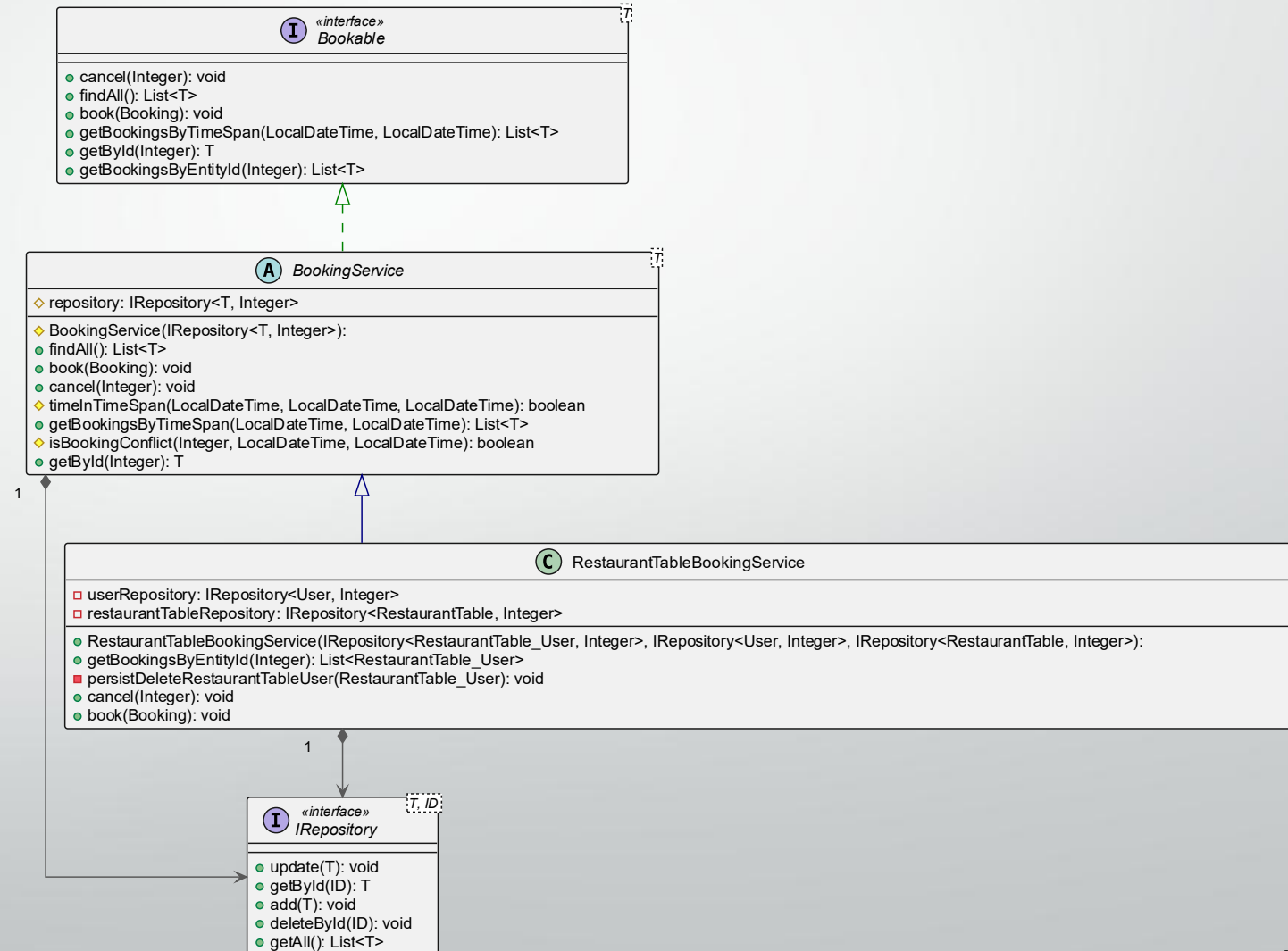


# DIP – Dependency Inversion Principle

Positivbeispiel –  
RestaurantTableBookingService

Warum erfüllt?

- IRepository ist ein Interface
- Konkrete Implementierungen werden über den Konstruktor injiziert
- Der Service weiß nicht, welche konkrete Datenbank oder Technik dahintersteckt

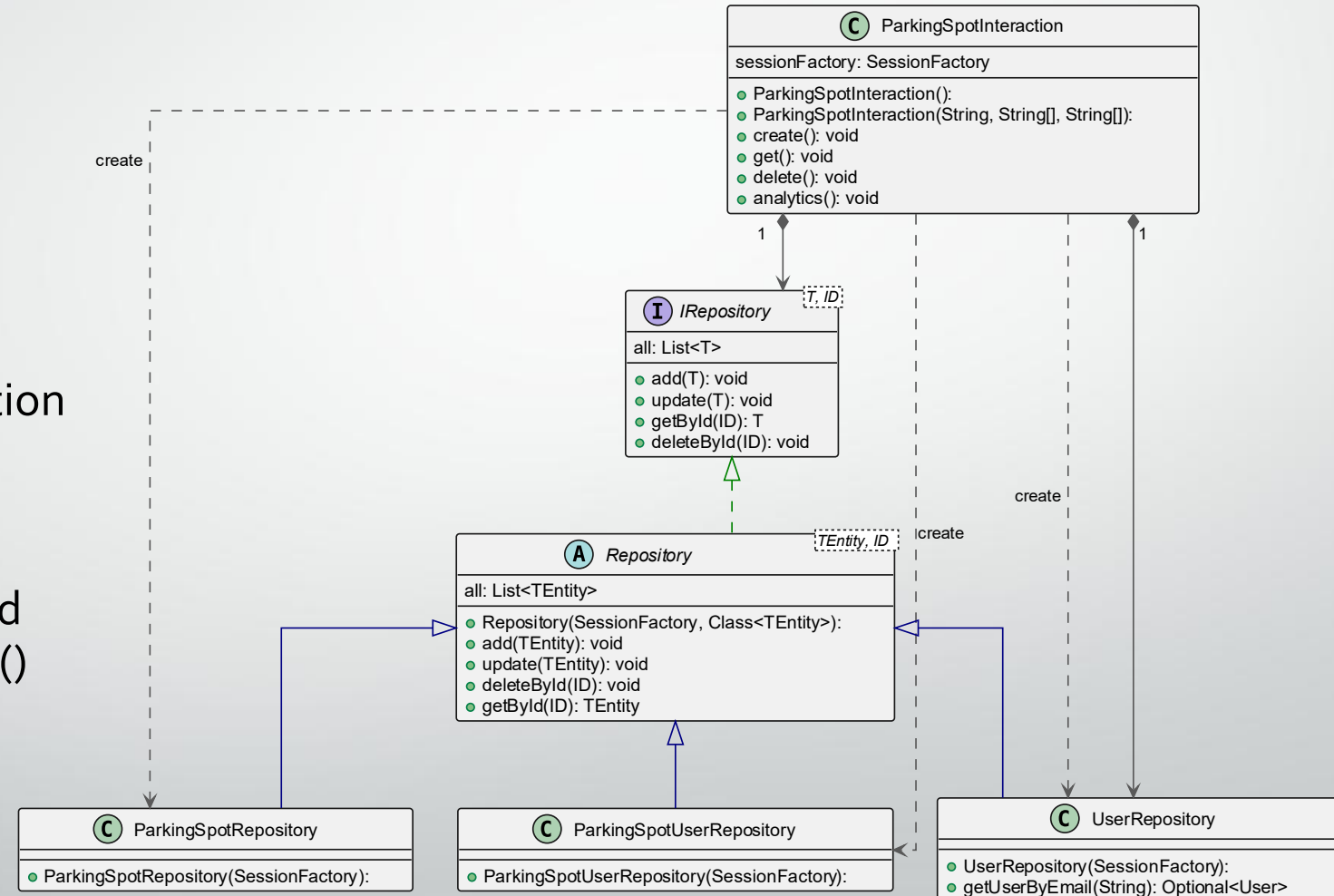


# DIP – Dependency Inversion Principle

## Negativbeispiel– ParkingSpotInteraction

Warum nicht erfüllt?

- ParkingSpotInteraction ruft direkt die implementierten Repositories auf (UserRepository) und erstellt sie über new()



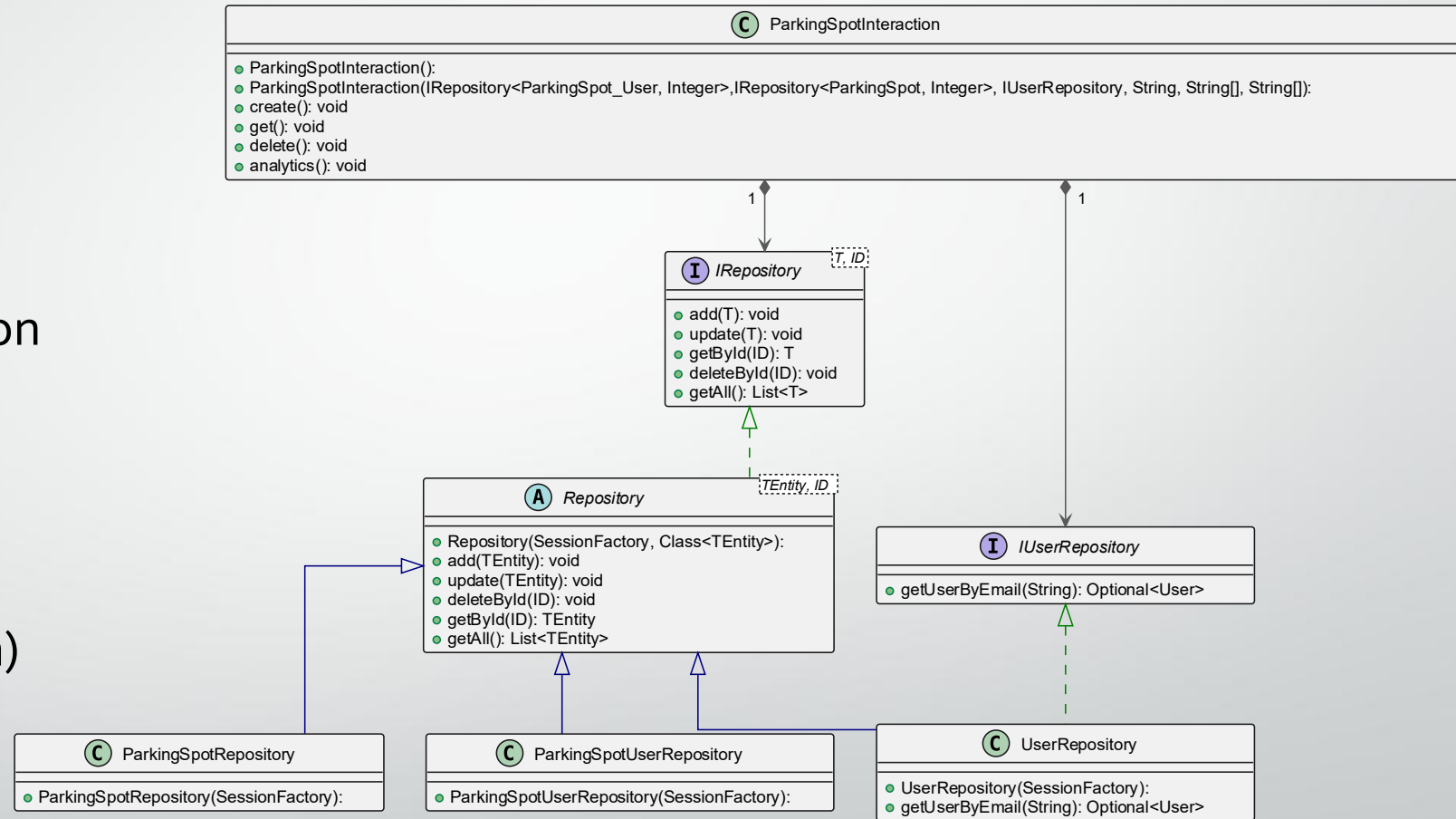
Anmerkung:  
Äquivalent zu den Repositoryaufrufen werden die Services aufgerufen

# DIP – Dependency Inversion Principle

## Negativbeispiel – ParkingSpotInteraction

### Verbesserung

- ParkingSpotInteraction kennt nur Interfaces
- Die konkreten Implementierungen werden nicht mehr selbst erstellt (Constructor Injection)



Anmerkung:  
Äquivalent zu den Repositoryaufrufen werden die Services über eine Constructorinjection übergeben

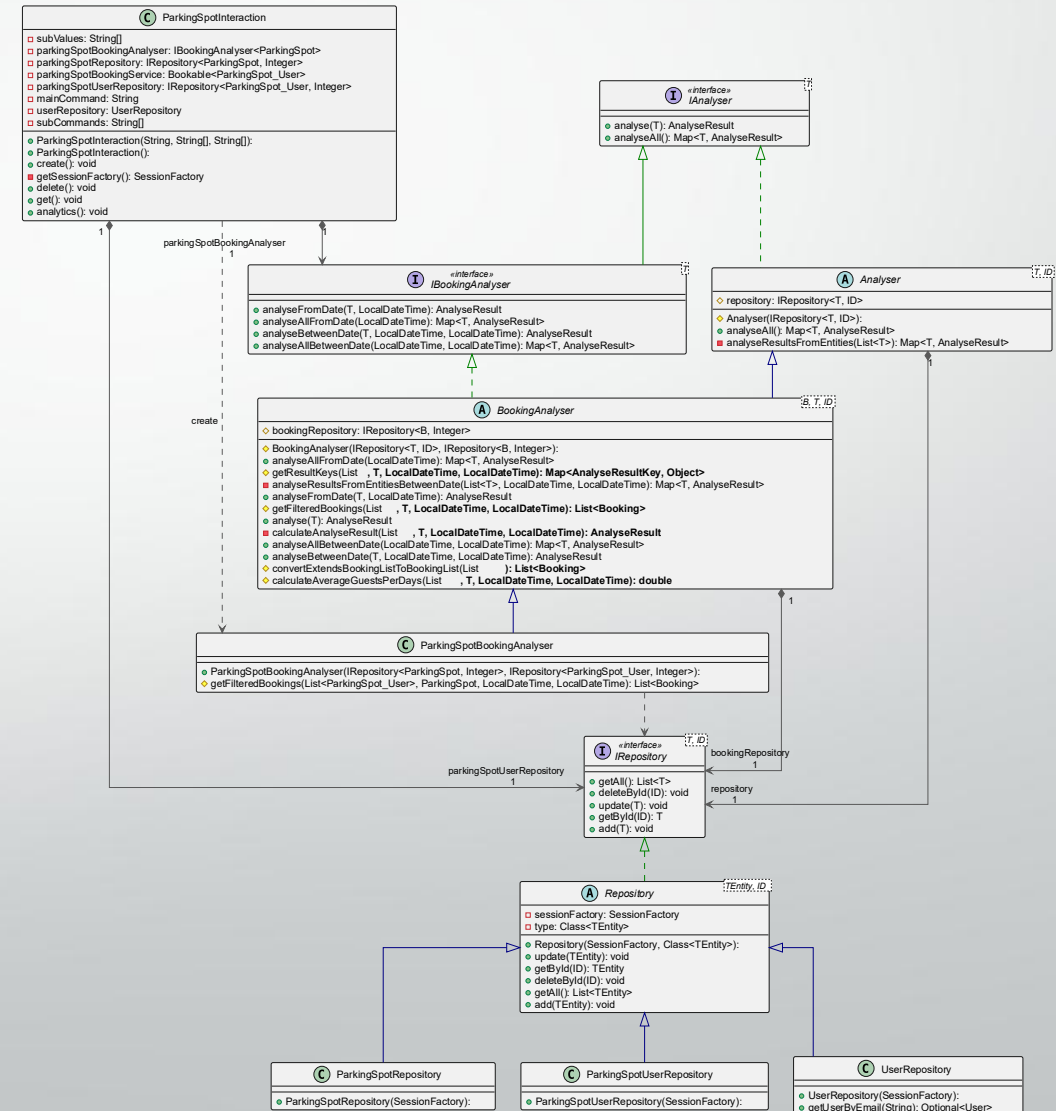


## Weitere Prinzipien

# GRASP – geringe Kopplung

## ParkingSpotBookingAnalyser

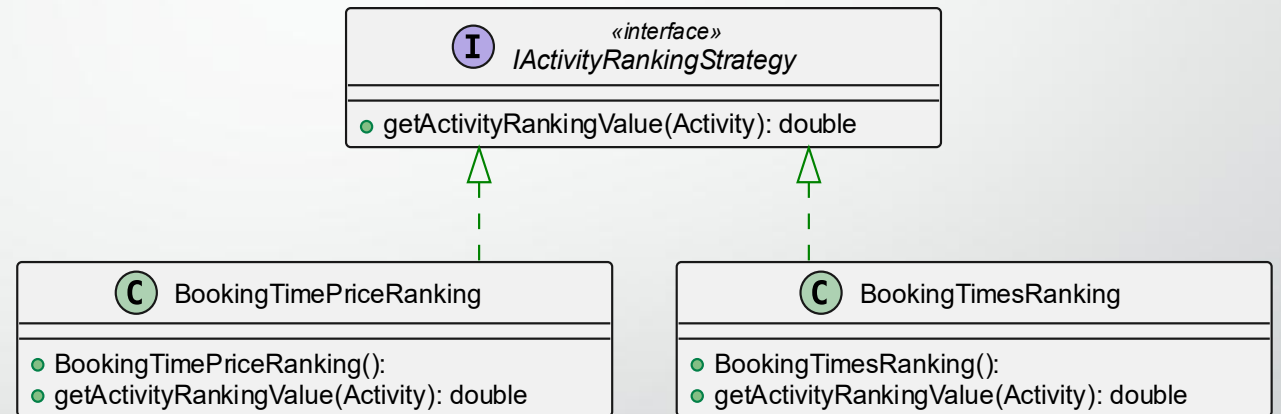
- Geringe Kopplung: BookingAnalyser und Unterklassen wie RoomBookingAnalyser arbeiten nur mit Abstraktionen (IRepository), nicht mit konkreten Implementierungen.



# GRASP – Polymorphismus

Begründung für den Einsatz

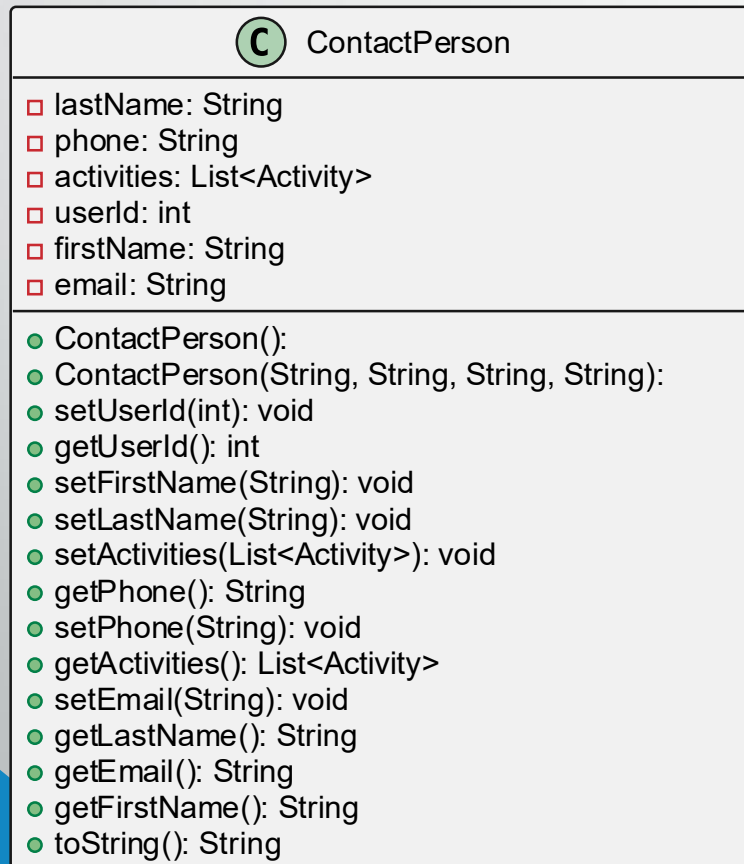
- Flexibilität
- Austauschbarkeit
- Erweiterbarkeit des Codes ohne Änderungen an den Nutzern des Interface.



# GRASP - DRY

ContactPerson und User haben überschneidende Attribute und Methoden

=> Auslagern in Person Klasse

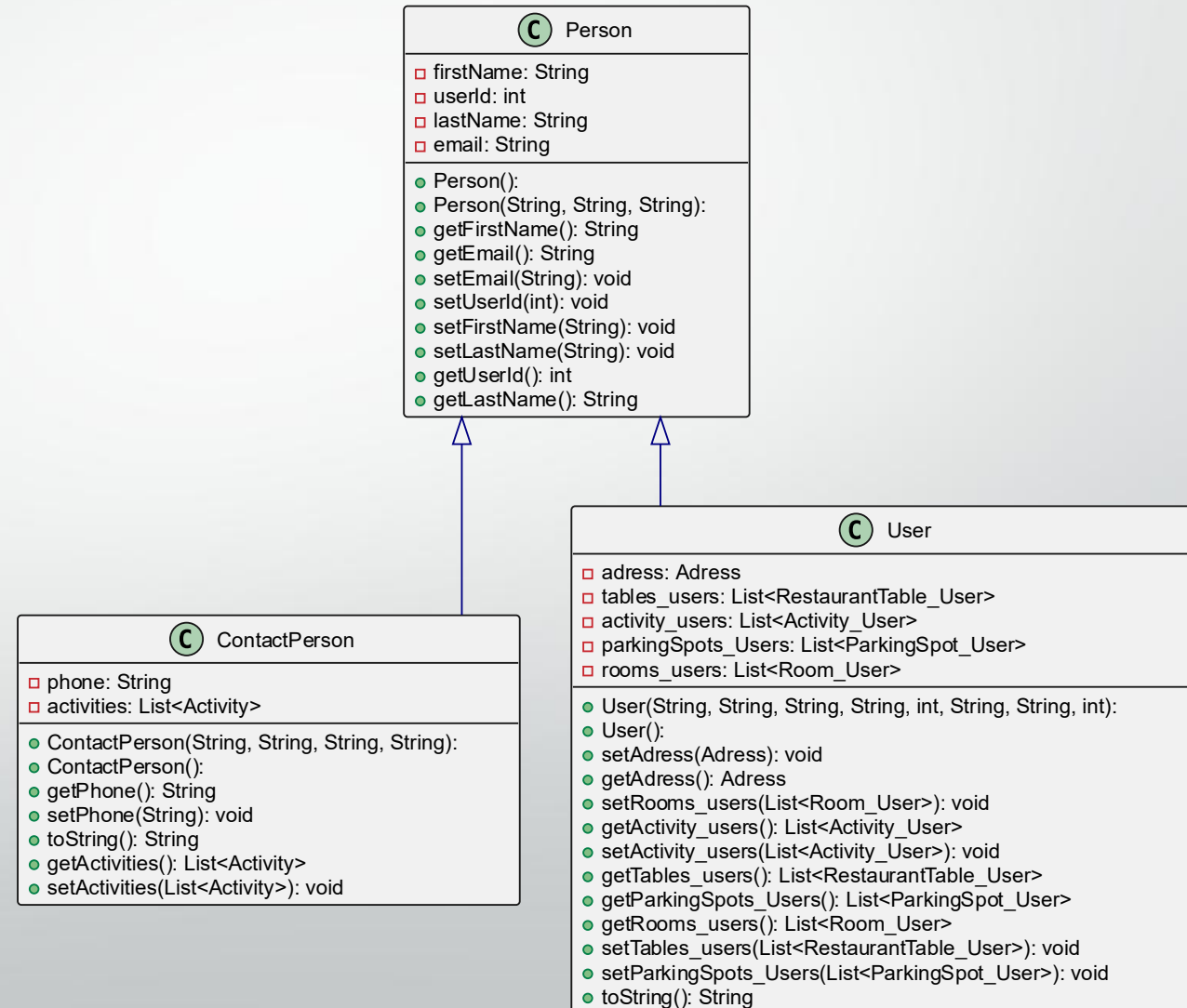


## Lösung - Auslagerung des duplizierten Codes in eigene Klasse


### Auswirkungen:

- Reduzierung der Redundanz
- Vereinfachung der Klasse User und ContactPerson
- Klarere Semantik, da ein User und eine ContactPerson eine Person sind

# GRASP - DRY







# Unit-Tests

# Unit Test - Überblick

- 28 Unit-Test
  - 4 Tests für die Überprüfung der Analysefunktion zu einer Buchung
  - 6 Tests für die Benutzerregistrierung & -verwaltung
  - 18 Tests für die Überprüfungen der Buchungen (erfolgreiche Buchung, Buchung mit Konflikt, Buchung stornieren, nicht existente Buchung stornieren) bei jedem Buchungsservice
    - ActivityBookingService
    - ParkingspotBookingService
    - RestaurantTableBookingService
    - RoomBookingService (+ 2 Tests für die kombinierte Buchung mit Parkingspot)

# UnitTest – ATRIP (1)

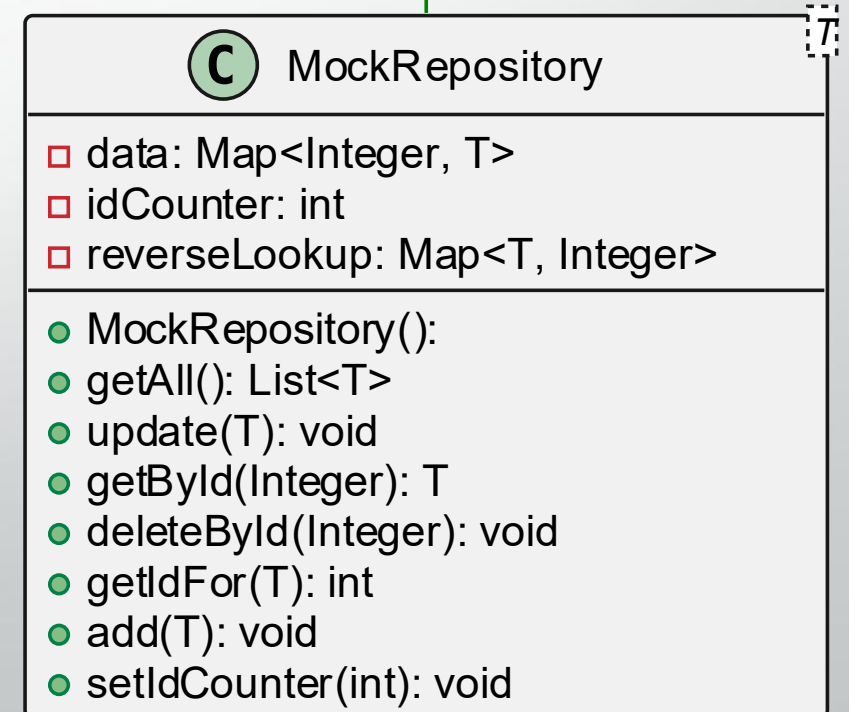
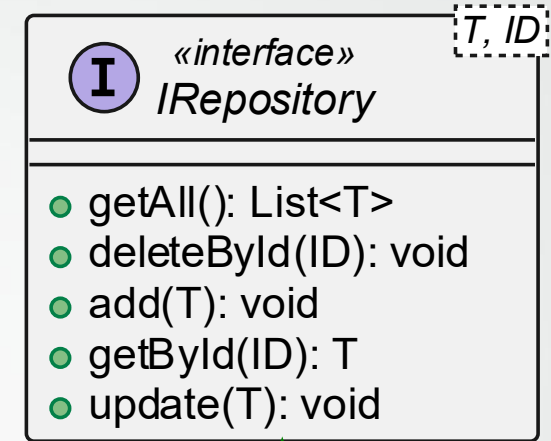
- Automatic
  - Alle Tests laufen ohne manuelles eingreifen statt und sie werden automatisch durch assertions validiert
  - Verwendung einer CI-Pipeline, welche bei jedem Push auf main das Programm baut und alle Tests durchläuft
- Thorough
  - 32% generelle Coverage, 82% Coverage bei den Services
  - Fokussierung auf die Geschäftslogik, die häufigsten Befehle (Nutzerverwaltung, buchen, stornieren, normale Analyse) werden getestet
    - Bei den Buchungen durch vorrausgehende Benutzereingaben werden auch die fehlerhafte Buchungen und fehlerhafte Stornierungen überprüft

# UnitTest – ATRIP (2)

- Professional
  - Verwendung von Polymorphismus um Codeduplikation zu vermeiden und Teststruktur der Servicestruktur anzupassen
    - Einfache Erweiterungen möglich
  - Selbstsprechende Methodennamen
  - Nachvollziehbare Codestruktur für ein einfaches Verständnis

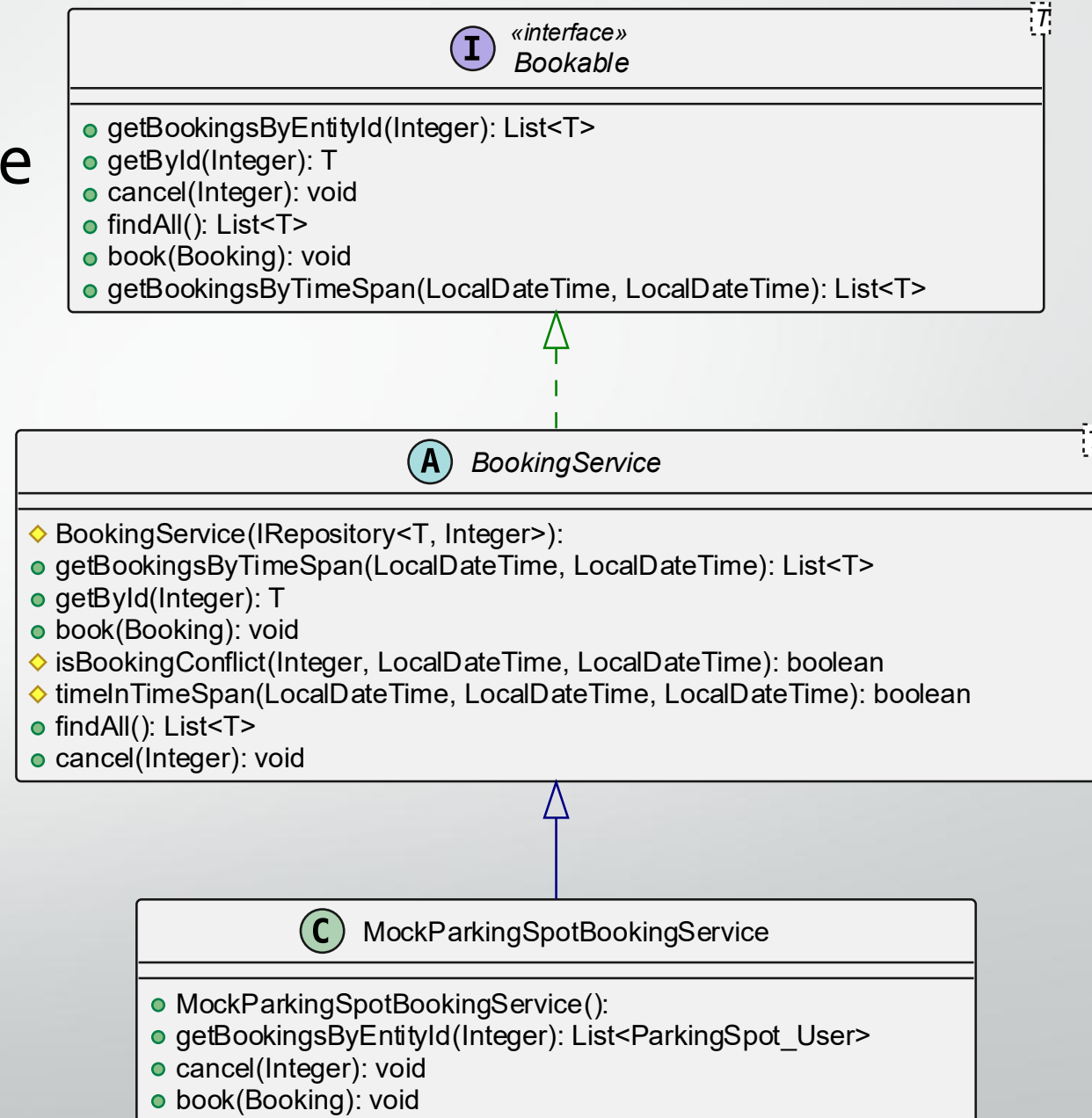
# Mocks - Repository

- Repositories mocken durch Implementierung des Repository-Interface
- Aufruf in jedem Test, um die Repositories der Entitäten zu mocken



# Mocks - ParkingspotBookingService

- Gemockter ParkingspotBookingService, da eine kombinierte Raumbuchung mit einem Parkplatz möglich ist. Dadurch wird eine isolierte Überprüfung der Raumbuchung gewährleistet
- Aufruf in RoomBookingServiceTest





# Domain Driven Design

# Domain Driven Design – Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
User	Repräsentiert einen Gast oder Kunden, der das System nutzt.	Zentraler Begriff, den sowohl Domänenexperten als auch Entwickler verwenden.
Room	Ein buchbares Hotelzimmer mit bestimmten Eigenschaften.	Wird direkt im fachlichen wie auch technischen Kontext identisch verwendet.
Activity	Freizeitangebot wie Spa, Fitness, Stadtführung etc., das gebucht werden kann.	Einheitlicher Begriff für buchbare Zusatzleistungen im System und in der Domäne.
RestaurantTable	Ein bestimmter Tisch im Hotelrestaurant, der reserviert werden kann.	Klare, gemeinsame Sprache zur Tischreservierung im Fachbereich und im Code.



# Repository - RoomRepository

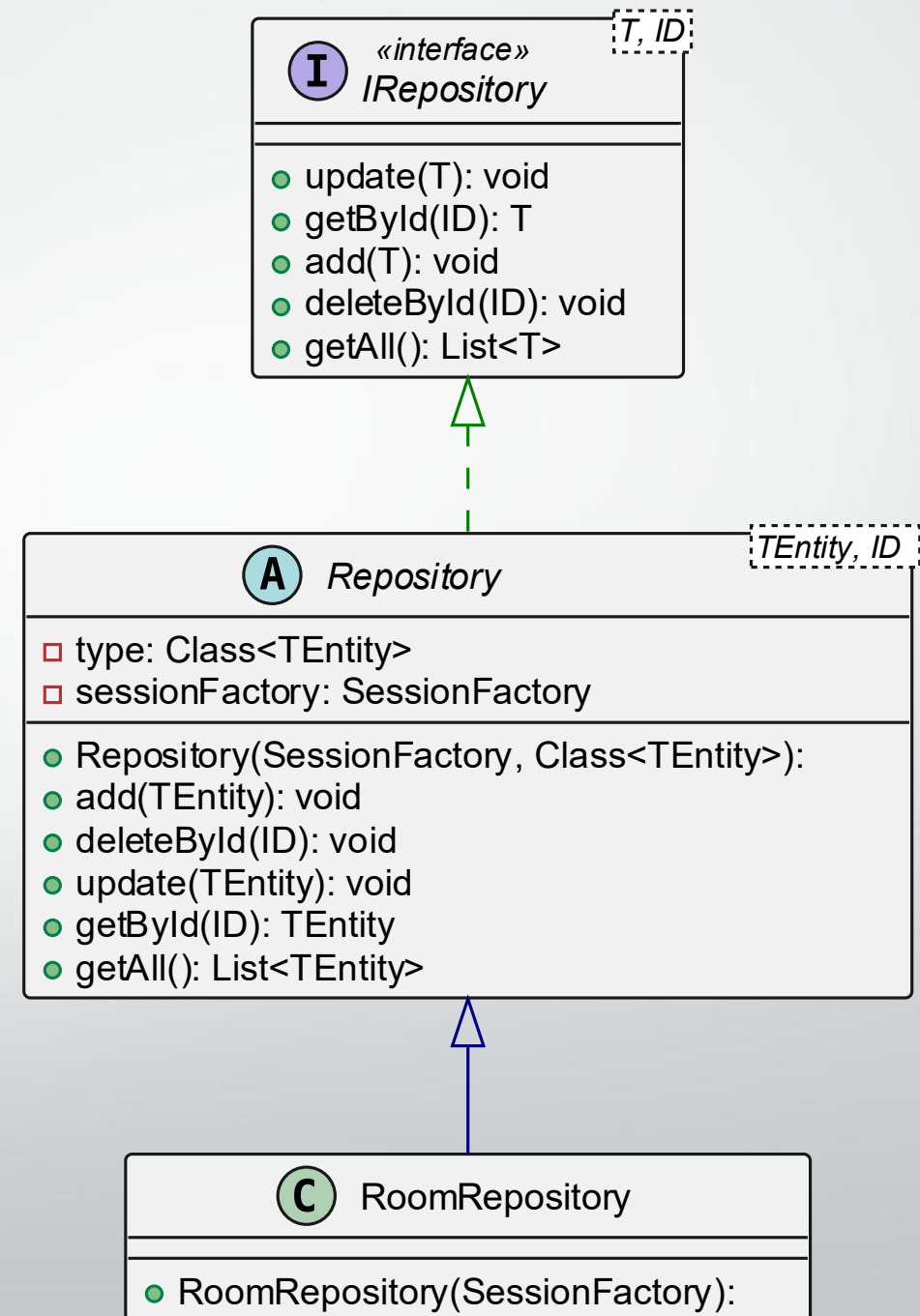
## **Beschreibung:**

RoomRepository erlaubt den Zugriff auf gespeicherte Daten über die CRUD-Funktionen

## **Begründung:**

Repositories trennen die Domäne von der Persistenz. Sie sind notwendig, um Räume aus einem Speicher zu laden, ohne dass die Domäne Details kennt.

# Repository - RoomRepository



# Aggregate – User & Address

## **Beschreibung:**

Ein User besitzt eigene Attribute wie E-Mail und eine eindeutige ID. Zusätzlich enthält er das Value Object Adresse, welches das Land, die Postleitzahl, den Ort, die Straße und die Hausnummer repräsentiert.

## **Begründung:**

Die UserId ist das einzige Mittel, um auf den User und damit auf das Value Object Adresse zuzugreifen, was den User zur Aggregat-Root macht. Darum bilden User und Adresse ein gemeinsames Aggregate.

# Aggregate- UML



# Entität - Raum

## **Beschreibung:**

Ein Raum stellt ein Zimmer im Hotel dar, welches über einen Zeitraum existiert und eine Identität hat (RoomId).

## **Begründung:**

Bei einem Raum handelt es sich um eine Entität, da er sich nicht über seine Werte definiert sondern über die Id. Die Werte sind über die Zeit veränderlich und ein Raum besitzt einen Lebenszyklus.

# Entität - Raum

## Room

- ❑ roomNumber: int
- ❑ doubleBeds: int
- ❑ description: String
- ❑ pricePerNight: int
- ❑ rooms\_users: List<Room\_User>
- ❑ singleBeds: int

- Room():
- Room(int, int, int, String, int):
- getRoomNumber(): int
- setRoomNumber(int): void
- setRooms\_users(List<Room\_User>): void
- getRooms\_users(): List<Room\_User>
- setSingleBeds(int): void
- setDoubleBeds(int): void
- getDoubleBeds(): int
- getSingleBeds(): int
- setDescription(String): void
- getDescription(): String
- getPricePerNight(): int
- setPricePerNight(int): void
- toString(): String

# Value Object - Adresse

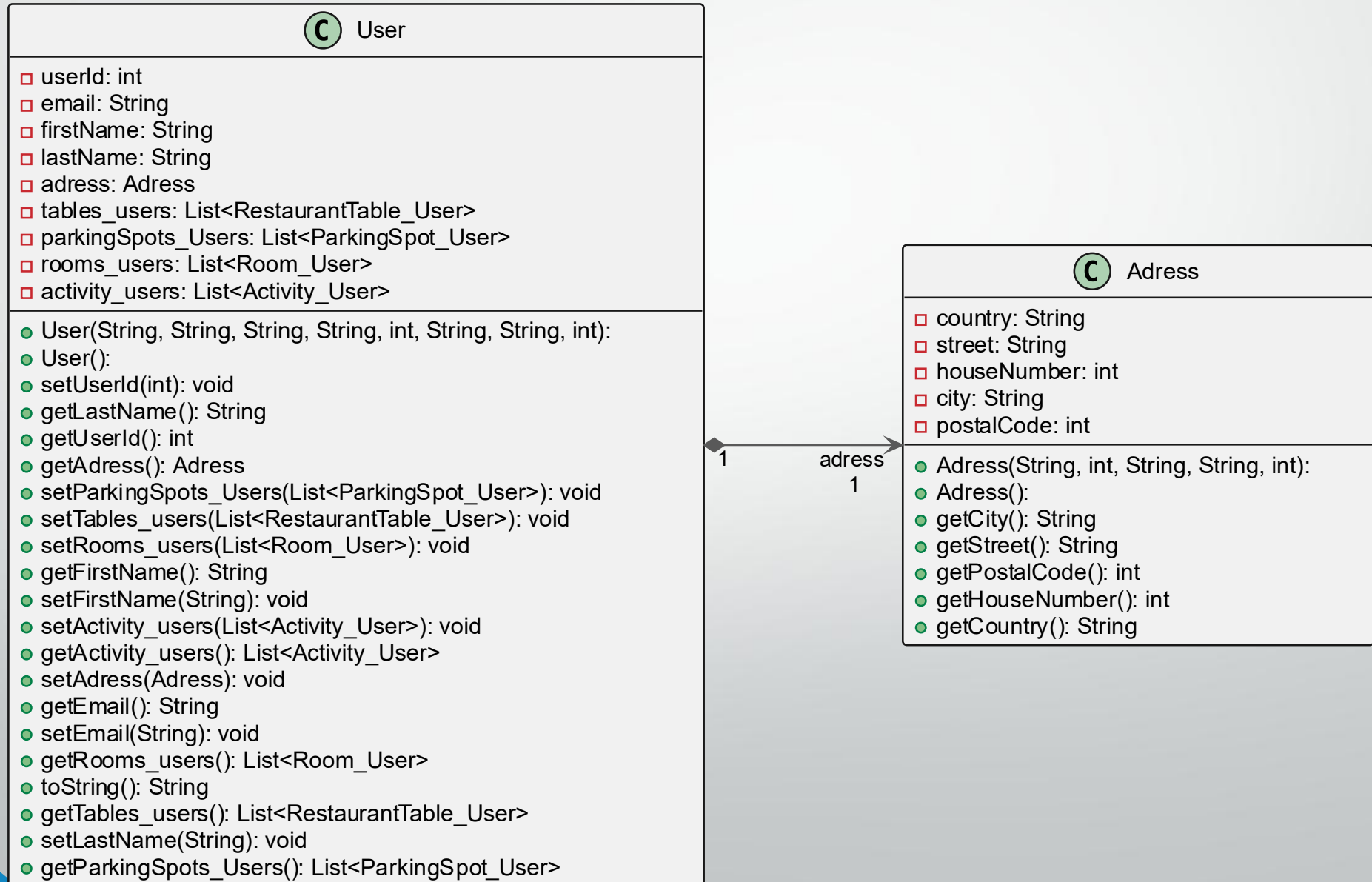
## **Beschreibung:**

In dem Value Object wird eine Adresse mit Land, Postleitzahl, Ort, Straße und Hausnummer gespeichert

## **Begründung:**

Adresse ist ein Value Object, da es keine eigene Identität hat und bei gleichen Parametern identisch ist.

# Value Object- Address







# Refactoring

# Refactoring

## Code Smells

- Long Method
- Switch Statement

## Refactoring

- replace Temp with Query
- rename method

# Code Smell – Long Method

```
protected Map<AnalyseResultKey, Object> getResultKeys(List<B> bookings,
                                                    T entity,
                                                    LocalDateTime startTime,
                                                    LocalDateTime endTime)
{
    Map<AnalyseResultKey, Object> resultKeys = new HashMap<>();

    LocalDateTime lastMonthBeginning = LocalDateTime.now().minusMonths(1).withDayOfMonth(1).withHour(0).withMinute(0).withSecond(0);
    LocalDateTime lastMonthEnding = YearMonth.now().minusMonths(1).atEndOfMonth().atTime(23, 59, 59, 999_999_999);
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_LAST_MONTH, calculateAverageGuestsPerDays(bookings, entity, lastMonthBeginning, lastMonthEnding));

    LocalDateTime thisMonthBeginning = LocalDateTime.now().withDayOfMonth(1).withHour(0).withMinute(0).withSecond(0);
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_THIS_MONTH, calculateAverageGuestsPerDays(bookings, entity, thisMonthBeginning, LocalDateTime.now()));

    LocalDateTime lastYearBeginning = LocalDateTime.now().minusYears(1).withDayOfYear(1).withHour(0).withMinute(0).withSecond(0);
    LocalDateTime lastYearEnding = LocalDateTime.of(LocalDateTime.now().getYear() - 1, 12, 31, 23, 59, 59, 999_999_999);
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_LAST_YEAR, calculateAverageGuestsPerDays(bookings, entity, lastYearBeginning, lastYearEnding));

    LocalDateTime thisYearBeginning = LocalDateTime.now().withDayOfYear(1).withHour(0).withMinute(0).withSecond(0);
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_THIS_YEAR, calculateAverageGuestsPerDays(bookings, entity, thisYearBeginning, LocalDateTime.now()));

    if (!startTime.isEqual(LocalDateTime.MIN))
    {
        resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_SELECTED_TIME, calculateAverageGuestsPerDays(bookings, entity, startTime, endTime));
    }

    return resultKeys;
}
```

# Code Smell – Long Method

## Aufgabe - Aktivität buchen

- über eine Methode werden die Parameter für eine Analyse berechnet inklusive der Zeitspannendefinition und der detaillierten Aufrufe der einzelnen Kennzahlen

## Probleme der GetResultKeys Methode

- Methode ist zu lang
- Methode ist unübersichtlich

## Lösung des Problems

- Methode in mehrere kleine Methoden aufteilen

# Code Smell – Long Method

```
protected Map<AnalyseResultKey, Object> getResultKeys(List<B> bookings,
                                                    T entity,
                                                    LocalDateTime startTime,
                                                    LocalDateTime endTime)
{
    Map<AnalyseResultKey, Object> resultKeys = new HashMap<>();

    addAverageGuestsLastMonth(resultKeys, bookings, entity);
    addAverageGuestsThisMonth(resultKeys, bookings, entity);
    addAverageGuestsLastYear(resultKeys, bookings, entity);
    addAverageGuestsThisYear(resultKeys, bookings, entity);
    addAverageGuestsSelectedTime(resultKeys, bookings, entity, startTime, endTime);

    return resultKeys;
}

private void addAverageGuestsLastMonth(Map<AnalyseResultKey, Object> resultKeys, List<B> bookings, T entity) {
    LocalDateTime start = getFirstDayOfLastMonth();
    LocalDateTime end = getLastDayOfLastMonth();
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_LAST_MONTH, calculateAverageGuestsPerDays(bookings, entity, start, end));
}

private void addAverageGuestsThisMonth(Map<AnalyseResultKey, Object> resultKeys, List<B> bookings, T entity) {
    LocalDateTime start = getFirstDayOfThisMonth();
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_THIS_MONTH, calculateAverageGuestsPerDays(bookings, entity, start, LocalDateTime.now()));
}

private void addAverageGuestsLastYear(Map<AnalyseResultKey, Object> resultKeys, List<B> bookings, T entity) {
    LocalDateTime start = getFirstDayOfLastYear();
    LocalDateTime end = getLastDayOfLastYear();
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_LAST_YEAR, calculateAverageGuestsPerDays(bookings, entity, start, end));
}

private void addAverageGuestsThisYear(Map<AnalyseResultKey, Object> resultKeys, List<B> bookings, T entity) {
    LocalDateTime start = getFirstDayOfThisYear();
    resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_THIS_YEAR, calculateAverageGuestsPerDays(bookings, entity, start, LocalDateTime.now()));
}

private void addAverageGuestsSelectedTime(Map<AnalyseResultKey, Object> resultKeys, List<B> bookings, T entity,
                                           LocalDateTime startTime, LocalDateTime endTime) {
    if (!startTime.isEqual(LocalDateTime.MIN)) {
        resultKeys.put(AnalyseResultKey.AVERAGE_BOOKING_GUESTS_PER_DAY_SELECTED_TIME, calculateAverageGuestsPerDays(bookings, entity, startTime, endTime));
    }
}
```

codesnap.dev

```
//Hilfsmethoden für Datumsermittlung
private LocalDateTime getFirstDayOfLastMonth() {
    return LocalDateTime.now().minusMonths(1).withDayOfMonth(1).withHour(0).withMinute(0).withSecond(0).withNano(0);
}

private LocalDateTime getLastDayOfLastMonth() {
    return YearMonth.now().minusMonths(1).atEndOfMonth().atTime(23, 59, 59, 999_999_999);
}

private LocalDateTime getFirstDayOfThisMonth() {
    return LocalDateTime.now().withDayOfMonth(1).withHour(0).withMinute(0).withSecond(0).withNano(0);
}

private LocalDateTime getFirstDayOfLastYear() {
    return LocalDateTime.now().minusYears(1).withDayOfYear(1).withHour(0).withMinute(0).withSecond(0).withNano(0);
}

private LocalDateTime getLastDayOfLastYear() {
    return LocalDateTime.of(LocalDateTime.now().getYear() - 1, 12, 31, 23, 59, 59, 999_999_999);
}

private LocalDateTime getFirstDayOfThisYear() {
    return LocalDateTime.now().withDayOfYear(1).withHour(0).withMinute(0).withSecond(0).withNano(0);
}
```

# Code Smell – Switch Statement

## Aufgabe Switch Statement

- Main Command überprüfen und korrekte Aktion aufrufen, um diese auszuführen

## Probleme Switch Statement

- Aufruf der Aktion immer ähnlich → Erzeugung von gleichen bzw. sehr ähnlichen Switch Statements

## Lösung des Problems

- Aufruf der Funktionalitäten in eigene Methoden in angelegten Klassen

# Vorher:

```
public class HotelBookingCLI
{
    ...
    public void execute()
    {
        ...
        MainCommands matchingCommand = MainCommands.findCommandsByValue(mainCommand);

        switch (MainCommands.findCommandsByValue(mainCommand))
        {
            case CREATE_USER:
                if (SubCommands.commandsPartOfEnum(subCommands) && MainCommands.checkSubCommandsMatchMainCommandGroup(this.mainCommand, subCommands))
                {
                    try (SessionFactory sessionFactory = HibernateUtil.getSessionFactory())
                    {
                        //get input data map
                        Map<SubCommands, String> extractedParameters = extractParameters(subValues);

                        IRepository<Room_User, Integer> roomUserRepository = new RoomUserRepository(sessionFactory);
                        IRepository<User, Integer> userRepository = new UserRepository(sessionFactory);
                        IRepository<Room, Integer> roomRepository = new RoomRepository(sessionFactory);
                        IRepository<ParkingSpot, Integer> parkingSpotRepository = new ParkingSpotRepository(sessionFactory);
                        IRepository<ParkingSpot_User, Integer> parkingSpotUserRepository = new ParkingSpotUserRepository(sessionFactory);

                        BookingService<ParkingSpot_User> parkingSpotBookingService = new ParkingSpotBookingService(
                            parkingSpotUserRepository,
                            userRepository,
                            parkingSpotRepository
                        );

                        RoomBookingService roomBookingService = new RoomBookingService(
                            roomUserRepository,
                            userRepository,
                            roomRepository,
                            parkingSpotBookingService
                        );

                        List<Room_User> roomUserList = roomBookingService.getBookingsByEntityId(Utils.createNumber(extractedParameters.get(SubCommands.ID)));

                        for (Room_User roomUser : roomUserList)
                        {
                            User currentUser = userRepository.getById(roomUser.getUser().getUserId());
                            Room currentRoom = roomRepository.getById(roomUser.getRoom().getRoomNumber());

                            System.out.println("Room booking information: " + currentUser.toString() + "\n" + currentRoom.toString());
                        }
                    }
                    HibernateUtil.shutdown();
                    break;
                }
            ...
        }
    }
}
```

- Viel ähnlicher Code in jeder Switch Abfrage
- Schlechte Lesbarkeit
- Nur schwer zu verstehen was gemacht wird
- Sehr schlechte Wartbarkeit
- Viel zu viel Code für ein Switch Statement

# Nachher:

```
public class HotelBookingCLI
{
    private String[] args;

    private UserInteraction userInteraction;
    private RoomBookingInteraction roomBookingInteraction;
    private ParkingSpotInteraction parkingSpotInteraction;
    private RestaurantTableInteraction restaurantTableInteraction;
    private ActivityBookingInteraction activityBookingInteraction;

    public void execute()
    {
        ...

        MainCommands matchingCommand = MainCommands.findCommandsByValue(mainCommand);

        switch (MainCommands.findCommandsByValue(mainCommand))
        {
            case CREATE_USER:
                this.userInteraction.create();
                break;
            case GET_USER:
                this.userInteraction.get();
                break;
            case DELETE_USER:
                this.userInteraction.delete();
                break;
            case CREATE_ROOM_BOOKING:
                this.roomBookingInteraction.create();
                break;
            ...
        }
    }
}
```

- Switch Statement aufgeräumter
- Besser Lesbarkeit & Strukturierung
- Eigene Klassen mit Methoden für einzelne Service Aufrufe



# Refactoring – Replace Temp with Query

Aktuell:

- BookingsTimesRanking

```
@Override
public double getActivityRankingValue(Activity activity)
{
    long bookingsLastYear = activity.getActivity_users().stream().filter(activityUser →
        activityUser.getStartDateTime().isAfter(LocalDate.now().minusYears(1))).count();
    long bookingsLastMonth = activity.getActivity_users().stream().filter(activityUser →
        activityUser.getStartDateTime().isAfter(LocalDate.now().minusMonths(1))).count();
    return bookingsLastYear + bookingsLastMonth * 2;
}
```

codesnap.dev



BookingTimesRanking

• BookingTimesRanking();  
• getActivityRankingValue(Activity): double

# Refactoring – Replace Temp with Query

Aktuell:

- Berechnung des Rankings in einer Methode mit zwei temporären Variablen

Problem:

- Eingeschränkte Lesbarkeit
- Berechnung der Buchungswerte nicht testbar

Lösung des Problems

- Methode aufsplitten

# Refactoring – Replace Temp with Query


Nachher:

```
@Override
public double getActivityRankingValue(Activity activity) {
    return countBookingsLastYear(activity) + countBookingsLastMonth(activity) * 2;
}

private long countBookingsLastYear(Activity activity) {
    return countBookingsSince(activity, LocalDateTime.now().minusYears(1));
}

private long countBookingsLastMonth(Activity activity) {
    return countBookingsSince(activity, LocalDateTime.now().minusMonths(1));
}

private long countBookingsSince(Activity activity, LocalDateTime since) {
    return activity.getActivityUsers().stream()
        .filter(activityUser -> activityUser.getStartDateTime().isAfter(since))
        .count();
}
```

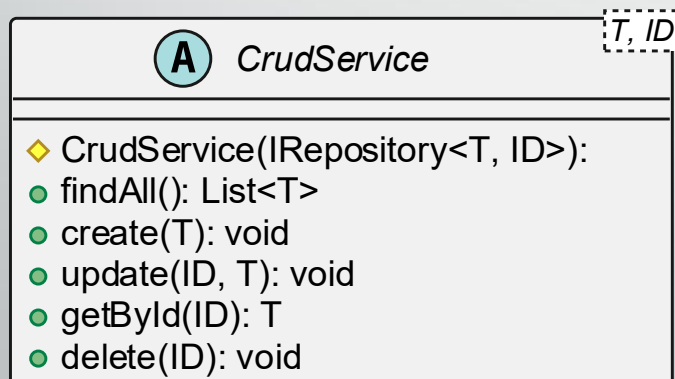
 BookingTimesRanking

- double getActivityRankingValue(Activity activity)
- long countBookingsLastYear(Activity activity)
- long countBookingsLastMonth(Activity activity)
- long countBookingsSince(Activity activity, LocalDateTime since)

# Refactoring – rename Method

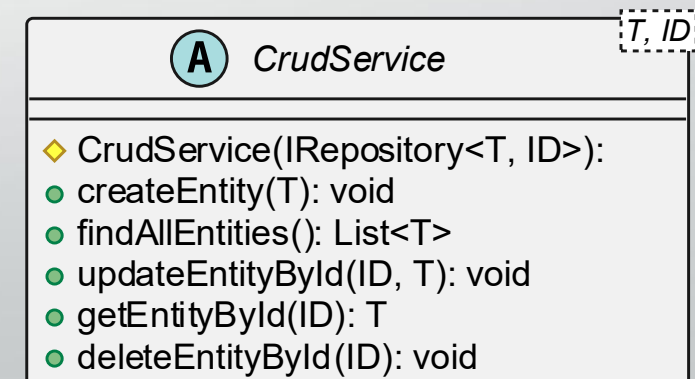
Aktuell:

- Unklar welches Objekt genau verändert werden soll und über welchen Schlüssel (basierend auf Methodennamen)



Refactored:

- Die Methodennamen wurden spezifiziert und vereinheitlicht (ById)





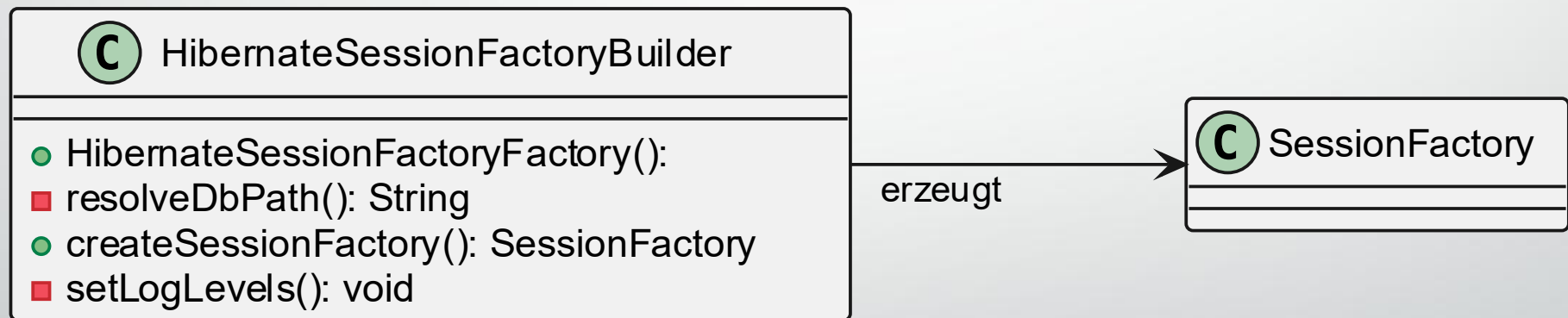
# Entwurfsmuster

# Builder - SessionFactoryBuilder

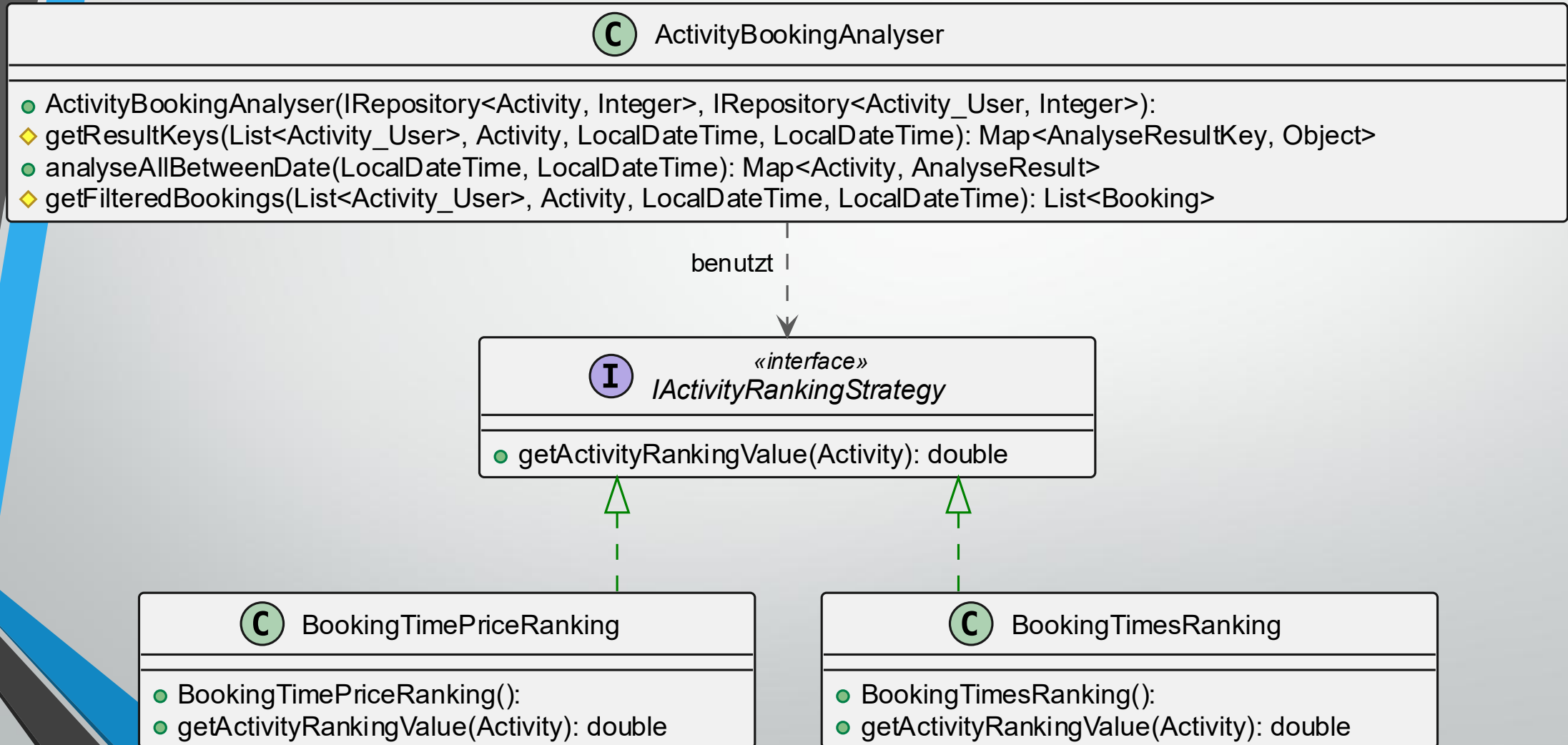
## Gründe

- Flexibilität
  - Einfache Änderung der Konfigurationen möglich bei einer Verwendung einer anderen Datenbank
- Testbarkeit
  - Durch die Möglichkeit der Erzeugung von neuen SessionFactory-Instanzen und einer Möglichkeit zu Überschreibung der Methoden, fällt das Testen leichter
- Lebenszyklus-Kontrolle
  - Genaue Entscheidung wann eine Instanz erstellt werden soll und wann sie wieder verworfen wird. Im Gegensatz zu Singeltons, welche meist früh erzeugt werden und lange leben

# Builder - SessionFactoryBuilder



# Strategy Pattern - ActivityRanking





# Strategy Pattern - ActivityRanking

## Gründe

- Vermeidung von übermäßigen if-Verzweigungen
- Veränderbare Auswahl zur Laufzeit ohne zugrundeliegende Implementierung zu verändern
- Erweiterbarkeit – Falls noch weitere Rankingstrategien umgesetzt werden sollen, können diese ohne großen Aufwand implementiert werden