****

**Hotel Management System**

**DESIGN PATTERNS (SE-408)**

**Abdul Haseeb(SE-16050)  
Bilal Ahmad (SE-16048)  
Hafiz Ahmed Abdullah (SE-16067)  
Syed Raffay Hussain (SE-16301)**

**INTRODUCTION**

**OBJECTIVE:**

A hotel management system that will facilitate the users and customers to easily book different types of rooms according to their needs. With a very flexible design to easily incorporate new rooms and other features

**PROJECT SCOPE:**

The scope of this project is to provide users and hotels with a simple yet flexible hotel management system. That a customer can use to reserve a room with ease and comfort.

**PROJECT PURPOSE:**

Use of different design patterns to build a real-world project application to light up the importance of Design patterns.

**SALIENT FEATURES**

* Allow users to reserve rooms through ease and comfort.
* Select a different kind of rooms according to your choice.
* Choose the different kinds of services and features that you need in your booked room.
* Payment right on the moment hiding all the complexity of payment features and giving users a very simple and clean interface.
* Allows Hotel Managers to easily add as many kinds of rooms as they want.
* Allows managers to add ad many different services in a specific room without breaking it.

**TECHNOLOGIES USED**

|  |  |
| --- | --- |
| **Operating System:** | **Windows 10** |
| **Language:** | **JAVA** |
| **IDE:** | **IntelliJ Idea** |

**DESIGN PATTERNS IMPLEMENTED**

Factory Pattern:

The factory Design pattern is being used in the project to create different kinds of rooms including Single bed or Double Bed. Moreover, you can easily add other different kinds of rooms in the future easily without breaking the code.

Facade Pattern:

The facade design pattern is being used to handle all the payment options. It provides a very simple User interface to the user with simple options including cash withdraw, balance check, etc. and hiding all the complexity behind the scenes.

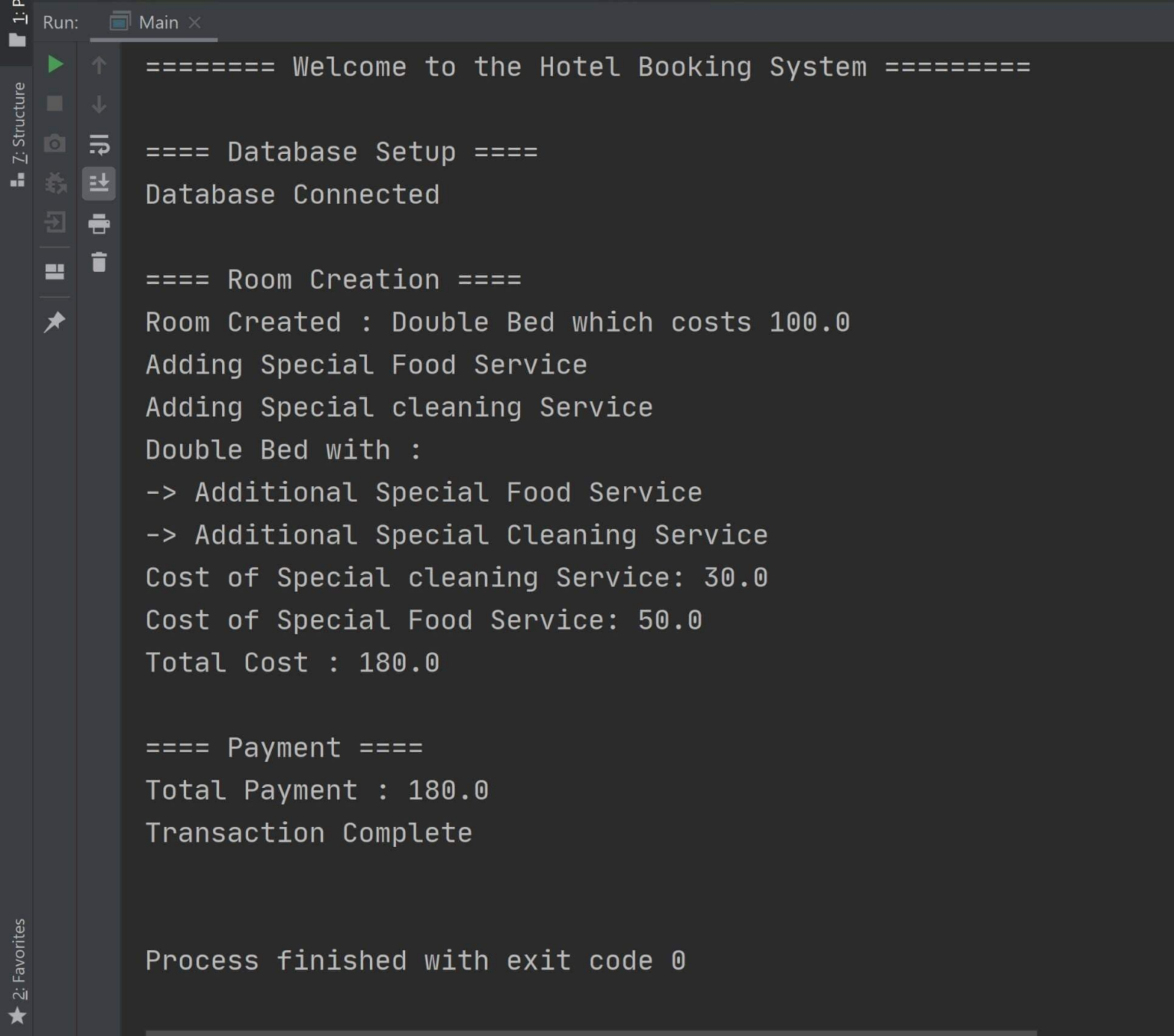
Decorator Pattern:

The Decorator pattern allows adding additional optional features and services to the room whether its a single bed or double bed. You can add additional food service or cleaning service if you want. Moreover, in the future, you can add other services too without breaking the code.

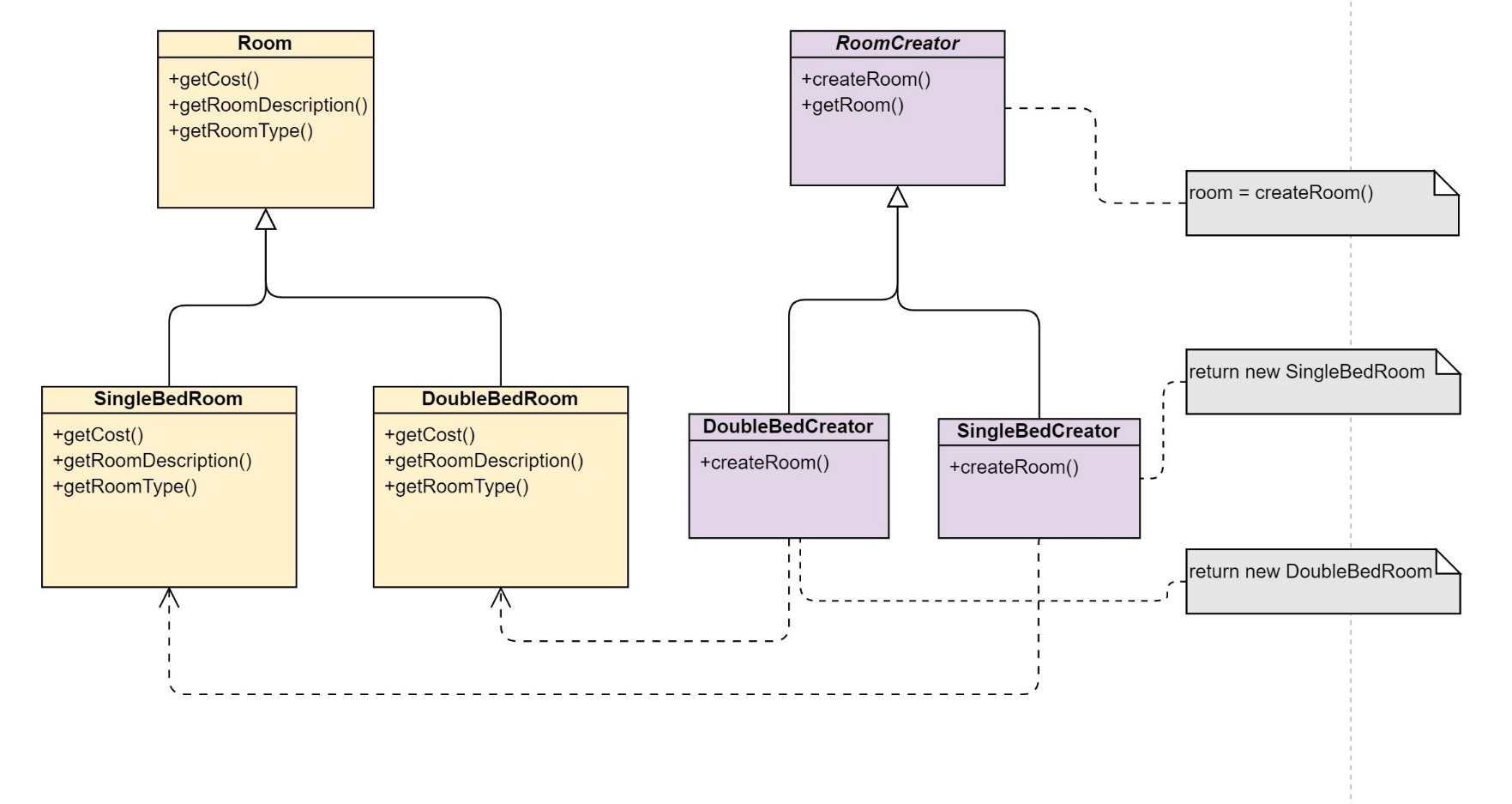
Singleton Pattern:

The Singleton pattern is used to keep a single instance of the database no matter from where ever you create database object it will be returning the same instance to ensure users only work with a single instance of the database.

**Project Output**

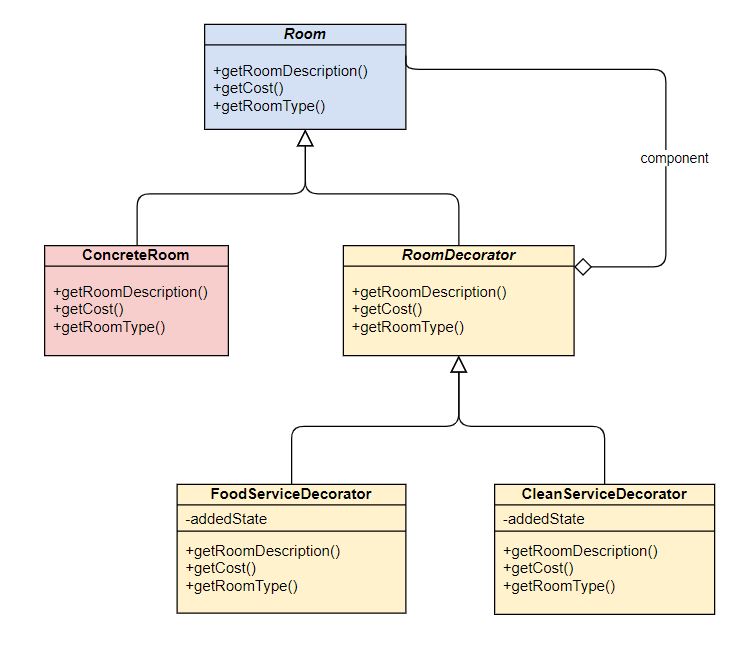
**CLASS DIAGRAMS**

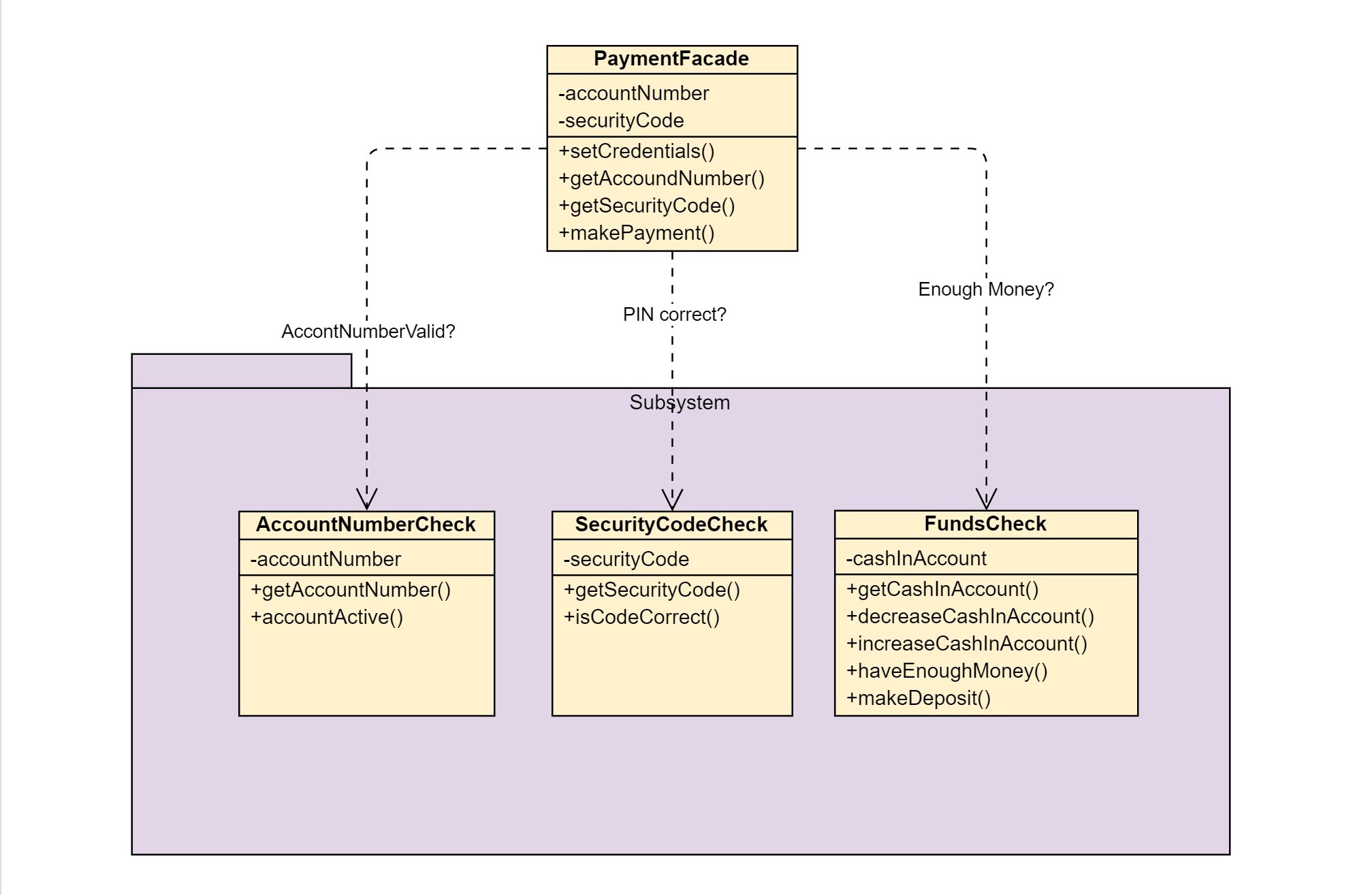
Factory Pattern:

****

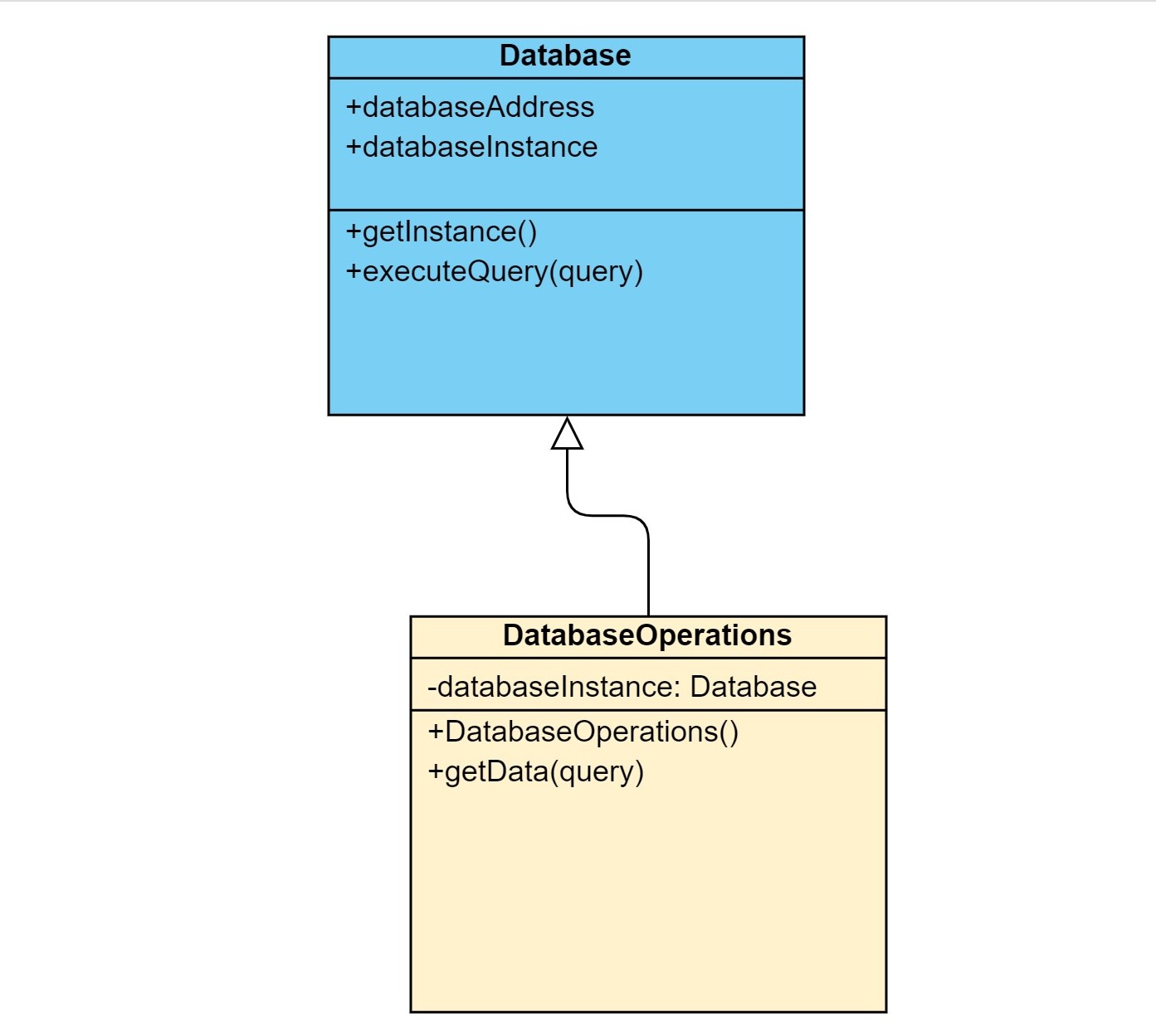
Facade Pattern

Decorator Pattern:



Facade Pattern:

Singleton Pattern:



**CODE**

|  |
| --- |
| **Main.java** |

package com.company;

import com.company.Decorator.CleanServiceDecorator;

import com.company.Decorator.FoodServiceDecorator;

import com.company.Facade.PaymentFacade;

import com.company.Factory.RoomFactory.DoubleBedCreator;

import com.company.Factory.RoomFactory.RoomCreator;

import com.company.Factory.RoomFactory.SingleBedCreator;

import com.company.Room.Room;

import com.company.Singleton.Database;

import com.company.Singleton.DatabaseOperations;

public class Main {

    private static *RoomCreator* roomCreator;

    public static *double* totalAmountToBePaid = 0;

    private static *Database* databaseInstance;

    private static *DatabaseOperations* databaseOperations;

    public static *void* main(*String*[] *args*) {

        System.out.println("\n\n======== Welcome to the Hotel Booking System =========\n");

        handleDatabaseConnection();

        handleRoomCreation(1); // 0 means singleBed, 1 means doubleBed

        handlePayment();

    }

    public static *void* handleRoomCreation(*int* *type*) {

        System.out.println("\n==== Room Creation ====");

        if (type == 0) {

            roomCreator = new SingleBedCreator();

        } else {

            roomCreator = new DoubleBedCreator();

        }

        System.out.println("Room Created : " + roomCreator.getRoom().getRoomType() + " which costs "

                + roomCreator.getRoom().getCost());

*Room* decoratedRoom = new CleanServiceDecorator(new FoodServiceDecorator(roomCreator.getRoom()));

        System.out.println(decoratedRoom.getRoomDescription());

        totalAmountToBePaid = decoratedRoom.getCost();

        System.out.println("Total Cost : " + totalAmountToBePaid);

    }

    public static *void* handlePayment() {

        System.out.println("\n==== Payment ====");

*PaymentFacade* payment = new PaymentFacade();

        payment.setCredentials(12345678, 1234);

        payment.makePayment(totalAmountToBePaid);

    }

    public static *void* handleDatabaseConnection() {

        System.out.println("==== Database Setup ====");

        databaseInstance = Database.getInstance("Database\_name");

        databaseOperations = new DatabaseOperations(databaseInstance);

        if (databaseInstance != null) {

            System.out.println("Database Connected");

        } else {

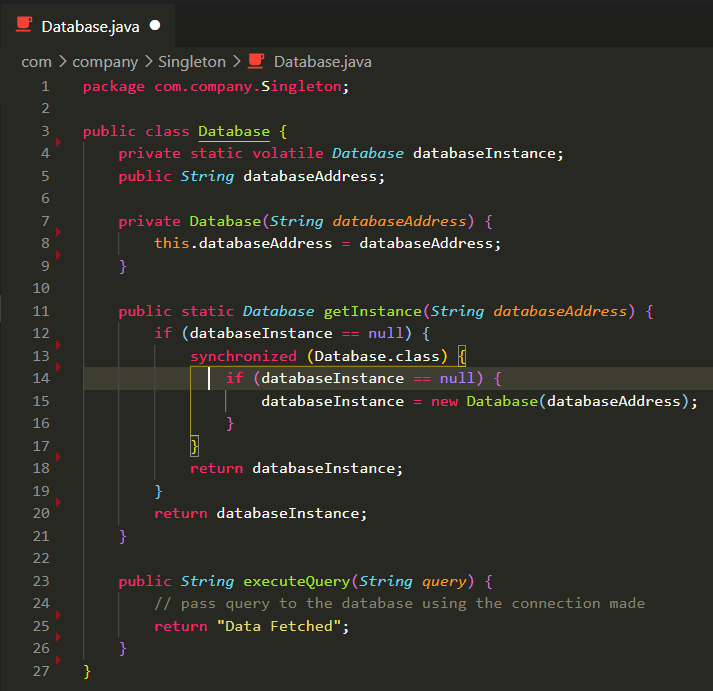
            System.out.println("Database Connection Failed");

        }

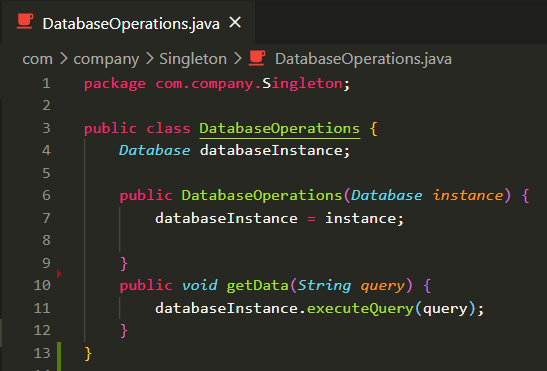
    }

}

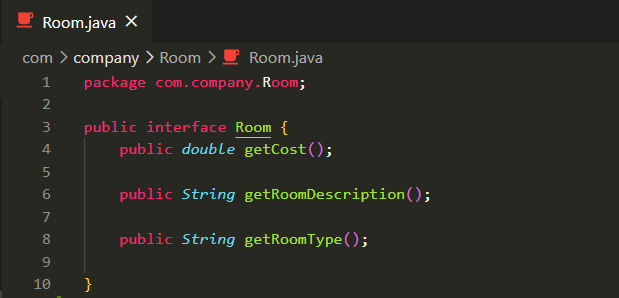
|  |
| --- |
| **Database.java** |



|  |
| --- |
| **DatabaseOperations.java** |



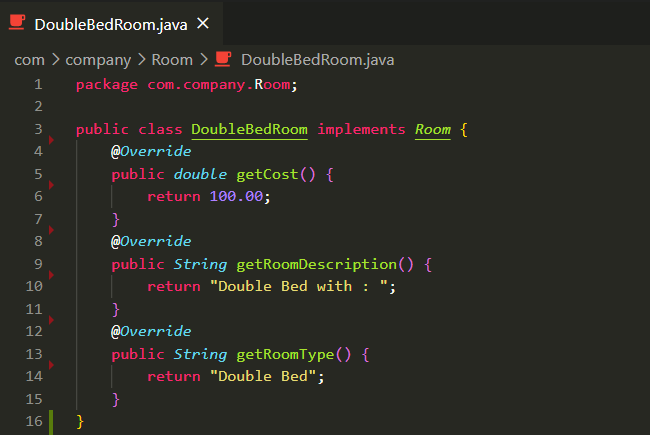
|  |
| --- |
| **Room.java** |



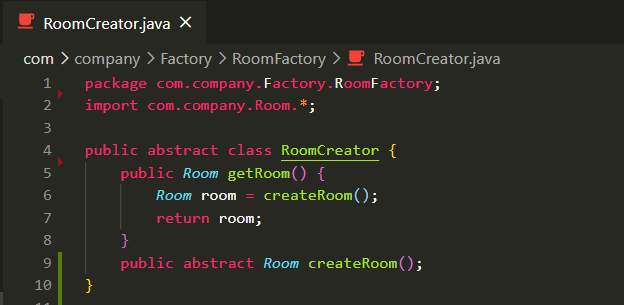
|  |
| --- |
| **SingleBedRoom.java** |



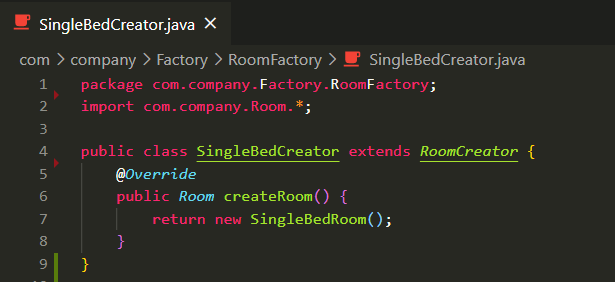
|  |
| --- |
| **DoubleBedRoom.java** |



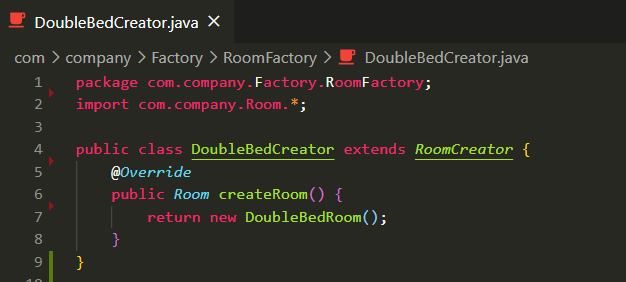
|  |
| --- |
| **RoomCreator.java** |



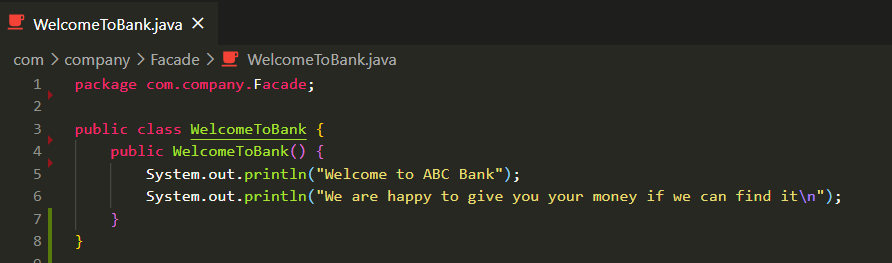
|  |
| --- |
| **SingleBedCreator.java** |



|  |
| --- |
| **DoubleBedCreator.java** |



|  |
| --- |
| **WelcomeToBank.java** |



|  |
| --- |
| **SecurityCodeCheck.java** |



|  |
| --- |
| **PaymentFacade.java** |

package com.company.Facade;

public class PaymentFacade {

    private *int* accountNumber;

    private *int* securityCode;

*AccountNumberCheck* acctChecker;

*SecurityCodeCheck* codeChecker;

*FundsCheck* fundChecker;

    public PaymentFacade() {

        acctChecker = new AccountNumberCheck();

        codeChecker = new SecurityCodeCheck();

        fundChecker = new FundsCheck();

    }

    public *void* setCredentials(*int* *newAcctNum*, *int* *newSecCode*) {

        accountNumber = newAcctNum;

        securityCode = newSecCode;

    }

    public *int* getAccountNumber() {

        return accountNumber;

    }

    public *int* getSecurityCode() {

        return securityCode;

    }

    public *void* withdrawCash(*double* *cashToGet*) {

        if (acctChecker.accountActive(getAccountNumber()) && codeChecker.isCodeCorrect(getSecurityCode())

                && fundChecker.haveEnoughMoney(cashToGet)) {

            System.out.println("Transaction Complete\n");

        } else {

            System.out.println("Transaction Failed\n");

        }

    }

    public *void* makePayment(*double* *payment*) {

        if (acctChecker.accountActive(getAccountNumber()) && codeChecker.isCodeCorrect(getSecurityCode())) {

            System.out.println("Total Payment : " + payment);

            fundChecker.makeDeposit(payment);

            System.out.println("Transaction Complete\n");

        } else {

            System.out.println("Transaction Failed\n");

        }

    }

    public *void* depositCash(*double* *cashToDeposit*) {

        if (acctChecker.accountActive(getAccountNumber()) && codeChecker.isCodeCorrect(getSecurityCode())) {

            fundChecker.makeDeposit(cashToDeposit);

            System.out.println("Transaction Complete\n");

        } else {

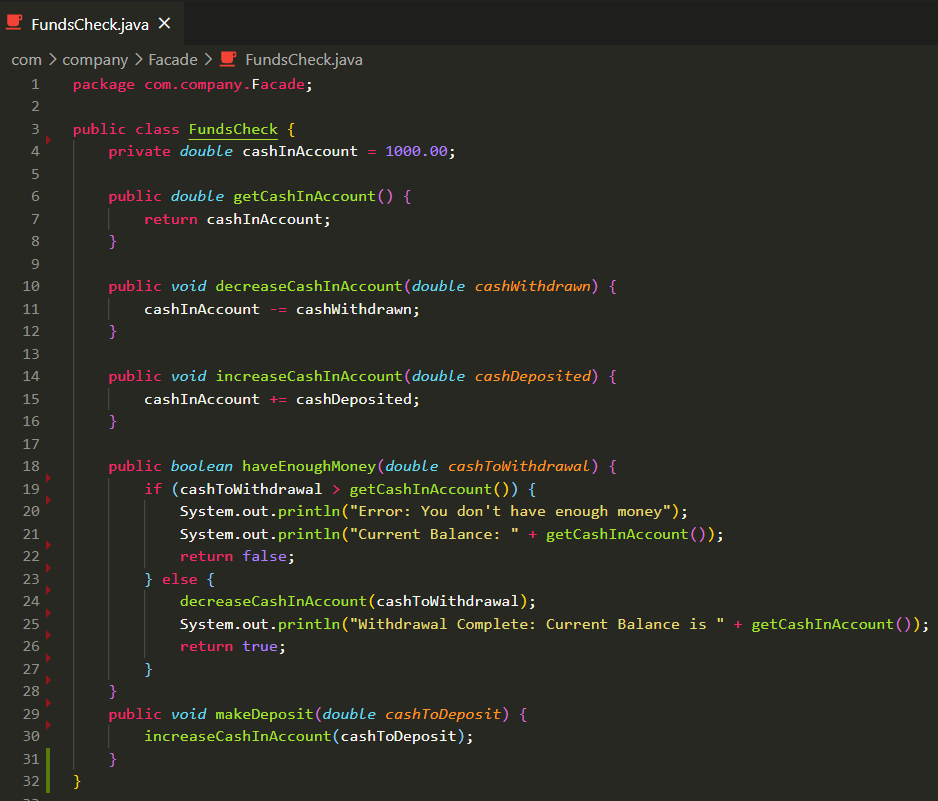
            System.out.println("Transaction Failed\n");

        }

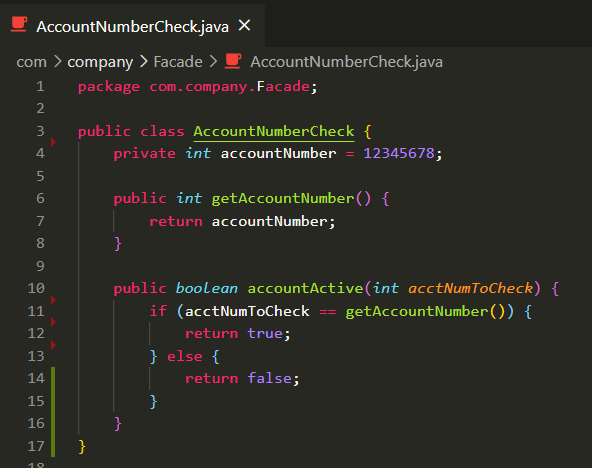
    }

}

|  |
| --- |
| **FundsCheck.java** |



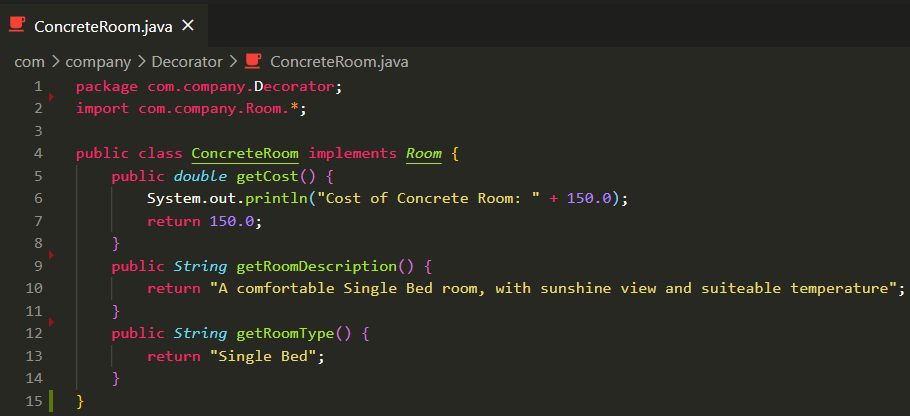
|  |
| --- |
| **AccountNumberCheck.java** |



|  |
| --- |
| **RoomDecorator.java** |



|  |
| --- |
| **ConcreteRom.java** |



|  |
| --- |
| **FoodServiceDecorator.java** |



|  |
| --- |
| **CleanServiceDecorator.java** |

