Translating UML into Code for

Master of Science

Information Technology

Bria Wright

University of Denver University College

April 6, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Implementing Parking System Application

This system is designed to manage parking operations for a university. The university has multiple parking lots with distinct fee structures. The typical customers at the university are faculty, staff, and students, who can register and obtain permits to park. Each permit is associated with a specific car and lot. The system needs to support operations such as customer registration, permit issuance, car entry, parking charge calculation based on duration, and transaction logging. Discounts must also apply to certain car types for example, compact cars, and the system must be modular, testable, and extendable. In the process of designing this Parking System Application, I am working in Eclipse IDE, Java Version 21, Junit Version: 5.9.3 and the Command Testing Class is MainClass.java.

Design

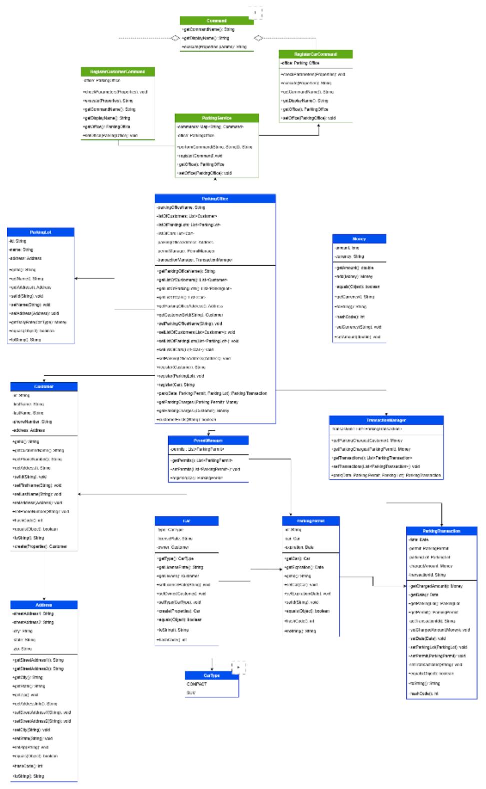


Figure 1: Class Diagram Module 1

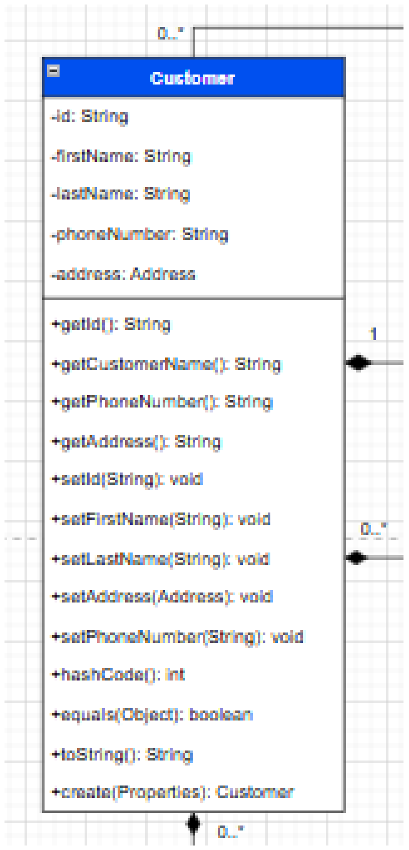


Figure 2: Class Diagram: Customer Class

Customer Class

This class represents a registered user of the parking system. Handles storing and retrieving customer data. It assigns a unique customer ID and ensures each customer's permits are managed cleanly. The multiplicity between Customer and ParkingPermit is *1..* (one or more permits per customer Stores identifying details and implements equals() and hashCode() to ensure correct behavior in hash-based collections as seen in Figure 2.

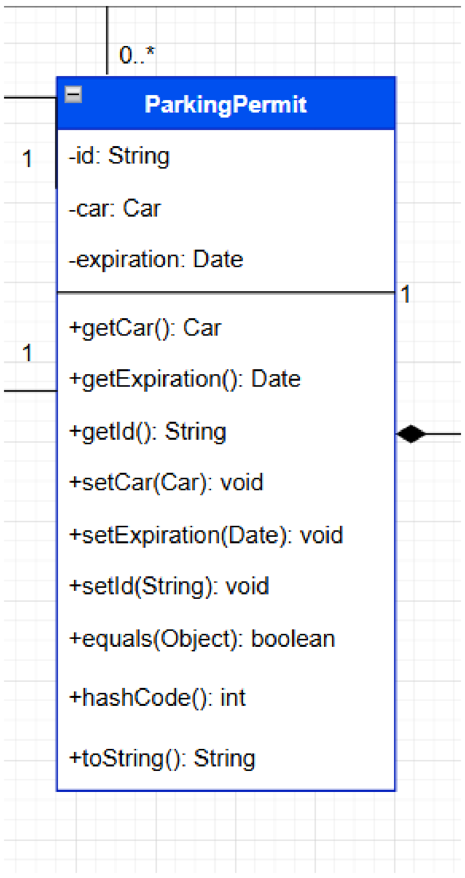


Figure 3: Class Diagram: ParkingPermit Class

ParkingPermit Class

The ParkingPermit class grants a customer the right to park a specific vehicle in designated parking lots. Each parking permit is linked to a Car and includes attributes like the permit ID and expiration date. The Car class is responsible for providing information about the vehicle, such as the make, model, and license plate number. Also, the ParkingPermit class has a many-to-one relationship with Car, as a car can have multiple parking permits associated with it over time, but each permit is linked to one specific car as shown in Figure 3.

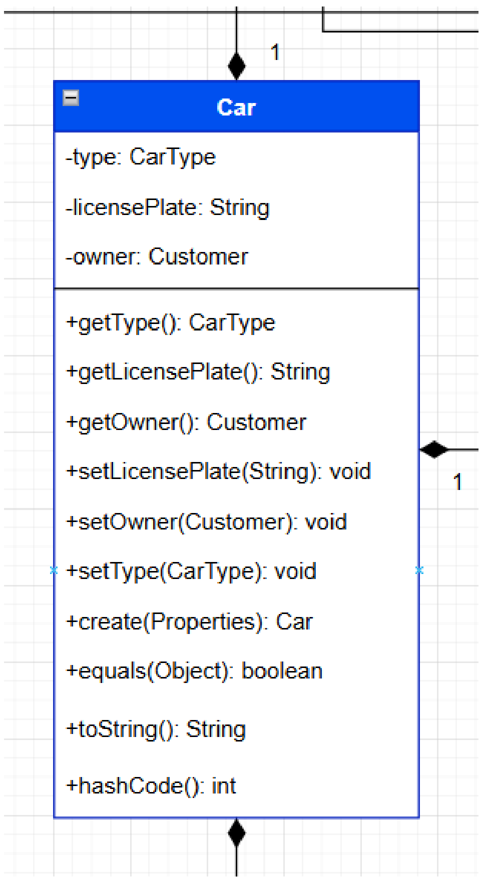


Figure 4: Class Diagram: Car Class

Car Class

The Car class holds information about the vehicle itself. Each car can be linked to a ParkingPermit, but a permit is not valid without being associated with a specific car. The Car class has attributes such as the car's make, model, license plate number, and registration details. As seen in Figure 4, there is a one-to-many relationship between Car and ParkingPermit, as one car can have multiple permits associated with it over time, but each permit corresponds to one car.

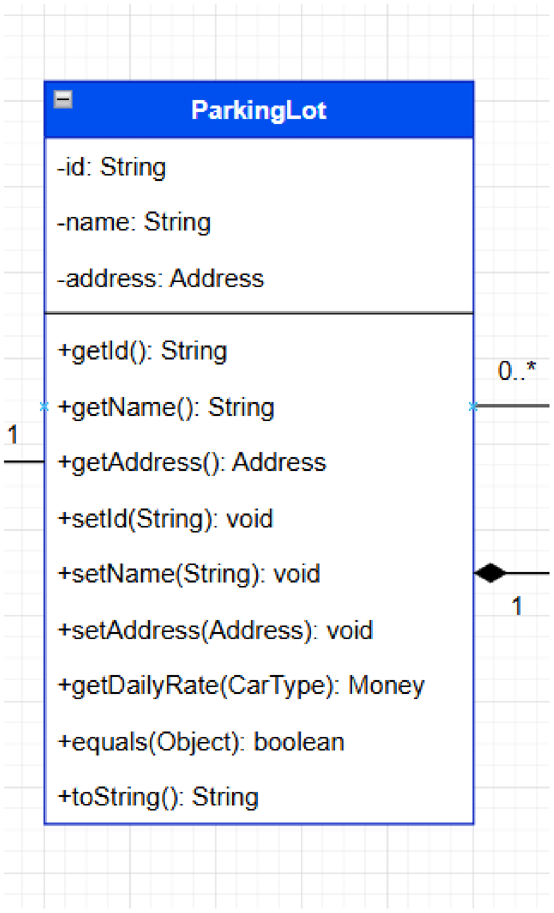


Figure 5: Class Diagram: ParkingLot Class

ParkingLot Class

In Figure 5, the ParkingLot class Stores name, hourly rate, and logic for calculating fees. Potential for extending different lots to have different rules. Parking lots also have varying pricing structures, which are essential in calculating the charges for parking. A ParkingLot can have many ParkingTransactions, indicating that multiple transactions can occur in the same parking lot. The multiplicity between ParkingLot and ParkingTransaction is 1.. (one or more transactions per parking lot).

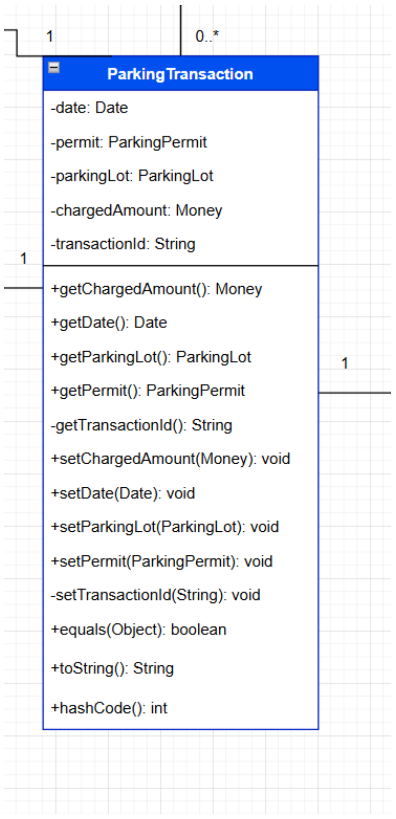

1
0..*
ParkingTransaction
-date: Date
-permit: ParkingPermit
-parkingLot: ParkingLot
-chargedAmount: Money
-transactionId: String
1
+getChargedAmount(): Money
+getDate(): Date
+getParkingLot(): ParkingLot
1
+getPermit(): ParkingPermit
-getTransactionId(): String
+setChargedAmount(Money): void
+setDate(Date): void
+setParkingLot(ParkingLot): void
+setPermit(ParkingPermit): void
-setTransactionId(String): void
+equals(Object): boolean
+toString(): String
+hashCode(): int

Figure 6: Class Diagram: ParkingTransaction Class

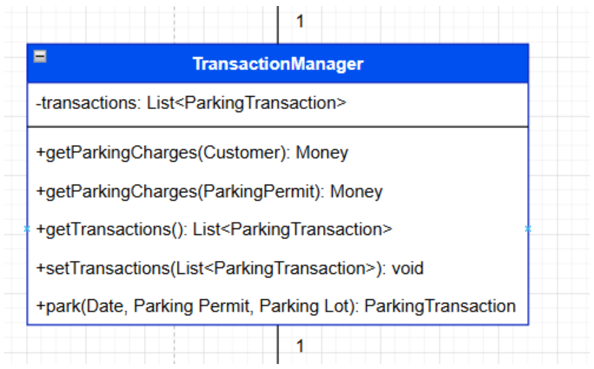

1
TransactionManager
-transactions: List<ParkingTransaction>
+getParkingCharges(Customer): Money
+getParkingCharges(ParkingPermit): Money
+getTransactions(): List<ParkingTransaction>
+setTransactions(List<ParkingTransaction>): void
+park(Date, Parking Permit, Parking Lot): ParkingTransaction
1

Figure 7: Class Diagram: TransactionManager Class

ParkingTransaction Class & TransactionManager Class

As seen in Figure 7, the TransactionManager and ParkingTransaction class log each parking event (start, end, fee). The TransactionManager ensures consistency and access to transaction records, while ParkingTransaction models each individual event.

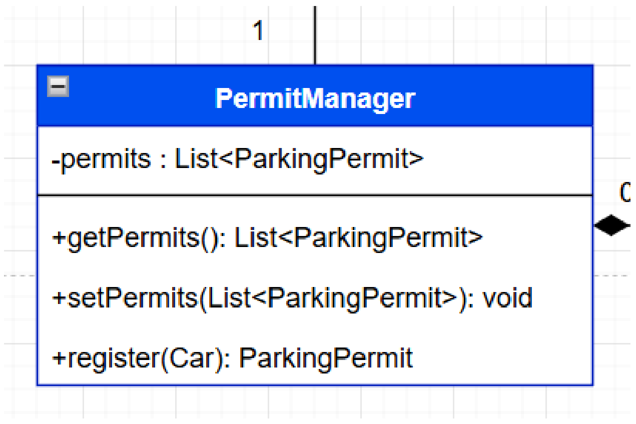

1
PermitManager
-permits : List<ParkingPermit>
+getPermits(): List<ParkingPermit>
+setPermits(List<ParkingPermit>): void
+register(Car): ParkingPermit

Figure 8: Class Diagram: PermitManager Class

PermitManager Class

The PermitManager class issues and retrieves permits. Each permit is uniquely identified and linked to a car and a parking lot. getPermitIds() allows querying all permits or filtering by customers. As seen in Figure 8, The PermitManager interacts with the ParkingPermit and Customer classes, ensuring that permits are correctly issued to customers and linked to their cars.

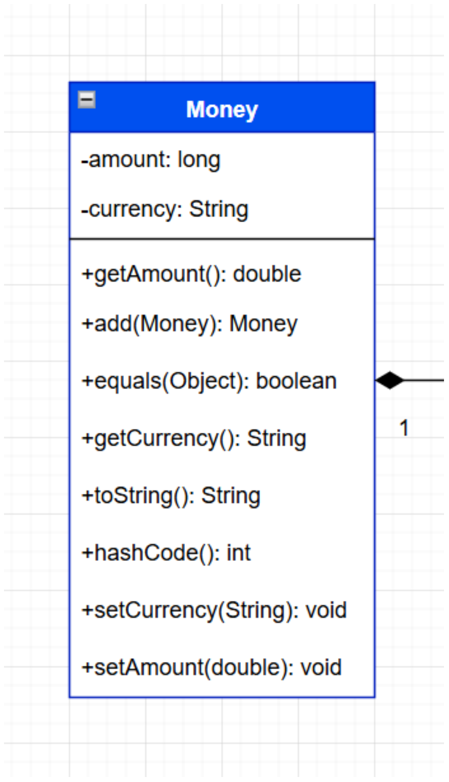


Figure 9: Class Diagram: Money Class

Money Class

In Figure 10, the Money class is a utility class that handles currency operations. Arithmetic operations (add, multiply) are wrapped to prevent rounding issues or logic bugs.

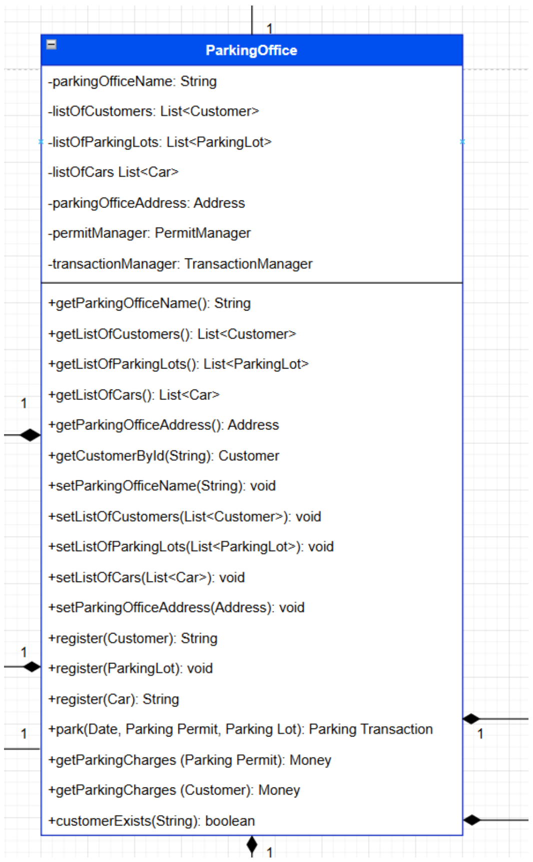

1
ParkingOffice
-parkingOfficeName: String
-listOfCustomers: List<Customer>
-listOfParkingLots: List<ParkingLot>
-listOfCars List<Car>
-parkingOfficeAddress: Address
-permitManager: PermitManager
-transactionManager: TransactionManager
+getParkingOfficeName(): String
+getListOfCustomers(): List<Customer>
+getListOfParkingLots(): List<ParkingLot>
+getListOfCars(): List<Car>
+getParkingOfficeAddress(): Address
+getCustomerById(String): Customer
+setParkingOfficeName(String): void
+setListOfCustomers(List<Customer>): void
+setListOfParkingLots(List<ParkingLot>): void
+setListOfCars(List<Car>): void
+setParkingOfficeAddress(Address): void
+register(Customer): String
+register(ParkingLot): void
+register(Car): String
+park(Date, Parking Permit, Parking Lot): Parking Transaction
+getParkingCharges (Parking Permit): Money
+getParkingCharges (Customer): Money
+customerExists(String): boolean
1
1
1
1
1

Figure 10: Class Diagram: ParkingOffice Class

ParkingOffice Class

The ParkingOffice class serves as the central interface between customers, parking transactions, and the parking management system. As seen in Figure 10, it acts as the controller/coordinator. It orchestrates customer registration, car linking, permit issuing, and logs parking events. Uses helper managers for encapsulating lower-level operations.

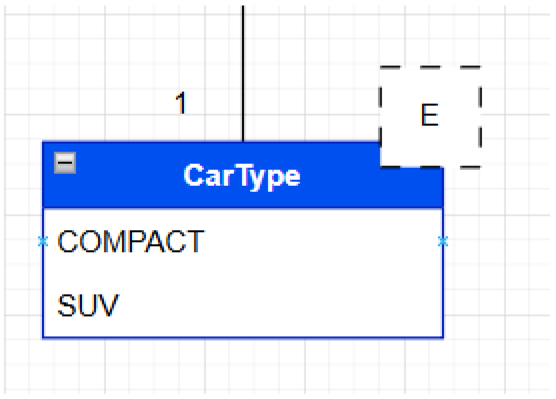


Figure 11: Class Diagram: CarType Enum Class

CarType Enum Class

In Figure 11, The CarType enum is used to classify different types of cars, in our case, the SUV and COMPACT cars. It provides a way to categorize cars for parking lot fee calculations, determining whether a car qualifies for any discounts or specific parking rules based on its type. There are no attributes directly in the enum, as it primarily defines a set of constants for the car types. The CarType enum helps in managing the different types of vehicles and determining how they interact with the parking lot's pricing structure.

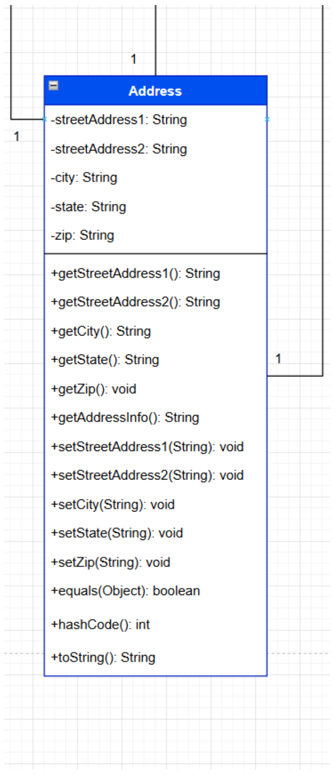


Figure 12: Class Diagram: Address Class

Address Class

In Figure 12, the Address class represents a customer's residential or business address, which is typically associated with their Customer profile. It ensures that customer location data is accurately represented and maintained. The Address class helps to manage and maintain customer location data in a structured way, making it easier to update or validate addresses in the parking system.

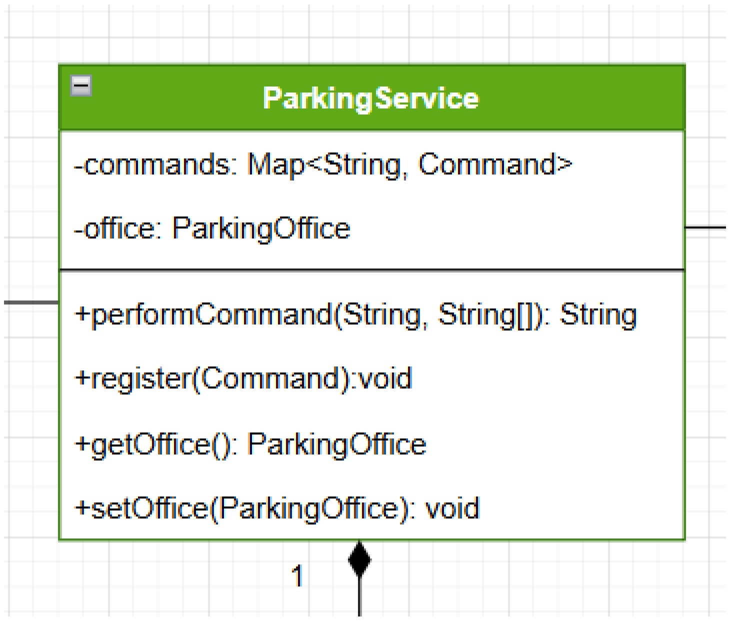

ParkingService
-commands: Map<String, Command>
-office: ParkingOffice
+performCommand(String, String[]): String
+register(Command):void
+getOffice(): ParkingOffice
+setOffice(ParkingOffice): void
1

Figure 13: Class Diagram: ParkingService Class

ParkingService Class

As seen in Figure 13, The ParkingService class serves as a simple command dispatcher. Provides a single point of entry for all high-level system actions. Maintains a registry of named command objects. Dispatches requests to the appropriate command using handle (commandName, properties). Simplifies future command integration and testing. Scalability: New commands can be added easily without altering existing logic.

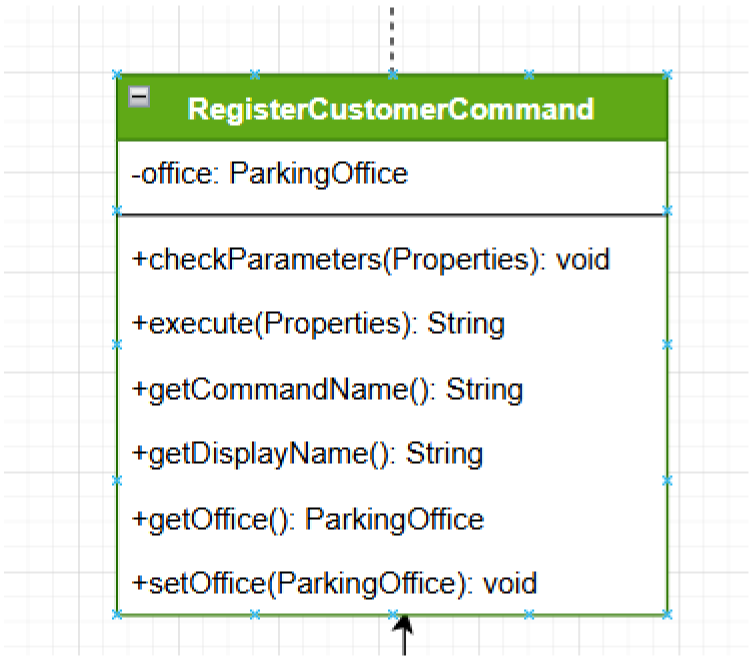

RegisterCustomerCommand
-office: ParkingOffice
+checkParameters(Properties): void
+execute(Properties): String
+getCommandName(): String
+getDisplayName(): String
+getOffice(): ParkingOffice
+setOffice(ParkingOffice): void

Figure 14: Class Diagram: RegisterCustomerCommand Class

RegisterCustomerCommand Class

As seen in Figure 14, The RegisterCustomerCommand class encapsulates the logic for creating and registering a new Customer into the system. It reads input data like first name, last name, phone, and address components from Properties. Then instantiates a Customer and registers it via ParkingOffice.register(Customer). Returns the generated customerId as confirmation. Integration: The class is used by ParkingService and in MainClass to handle customer onboarding. Also, it throws exceptions if required fields are missing or invalid.

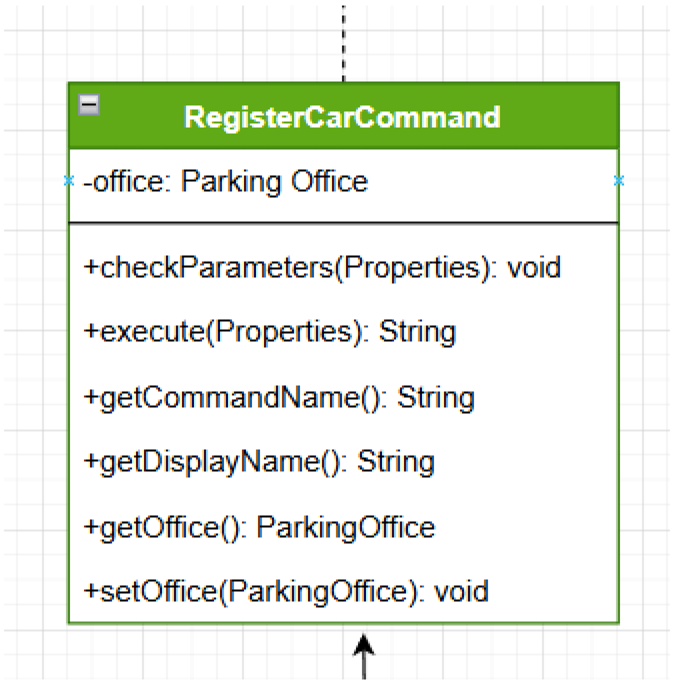

RegisterCarCommand
-office: Parking Office
+checkParameters(Properties): void
+execute(Properties): String
+getCommandName(): String
+getDisplayName(): String
+getOffice(): ParkingOffice
+setOffice(ParkingOffice): void

Figure 15: Class Diagram: RegisterCarCommand Class

RegisterCarCommand Class

As seen in Figure 15, The RegisterCarCommand class handles the complete process of registering a car for a customer and issuing a parking permit. Expects a valid customerId, licensePlate, and carType from Properties. Locates the customer in ParkingOffice, creates a Car, and registers both the car and a ParkingPermit. Returns the permit ID upon success. Validation: Verifies the customer's exist and inputs are valid. Error Handling: Provides descriptive errors for issues like unknown customers or invalid car types

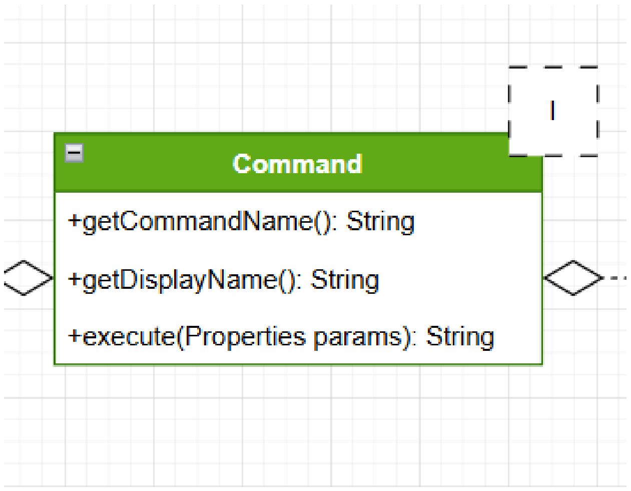

Command
+getCommandName(): String
+getDisplayName(): String
+execute(Properties params): String

Figure 16: Class Diagram: Command Interface Class

Command Class

As seen in Figure 16, The *Command* class acts as the contract for all executable user or system operations. It behaves with its own logic for handling operations through the execute(Properties) method. It enables command objects to be passed, stored, and reused dynamically. Lastly, it encourages loose coupling and future extensibility for example, ParkCarCommand, CalculateChargesCommand could be added later.

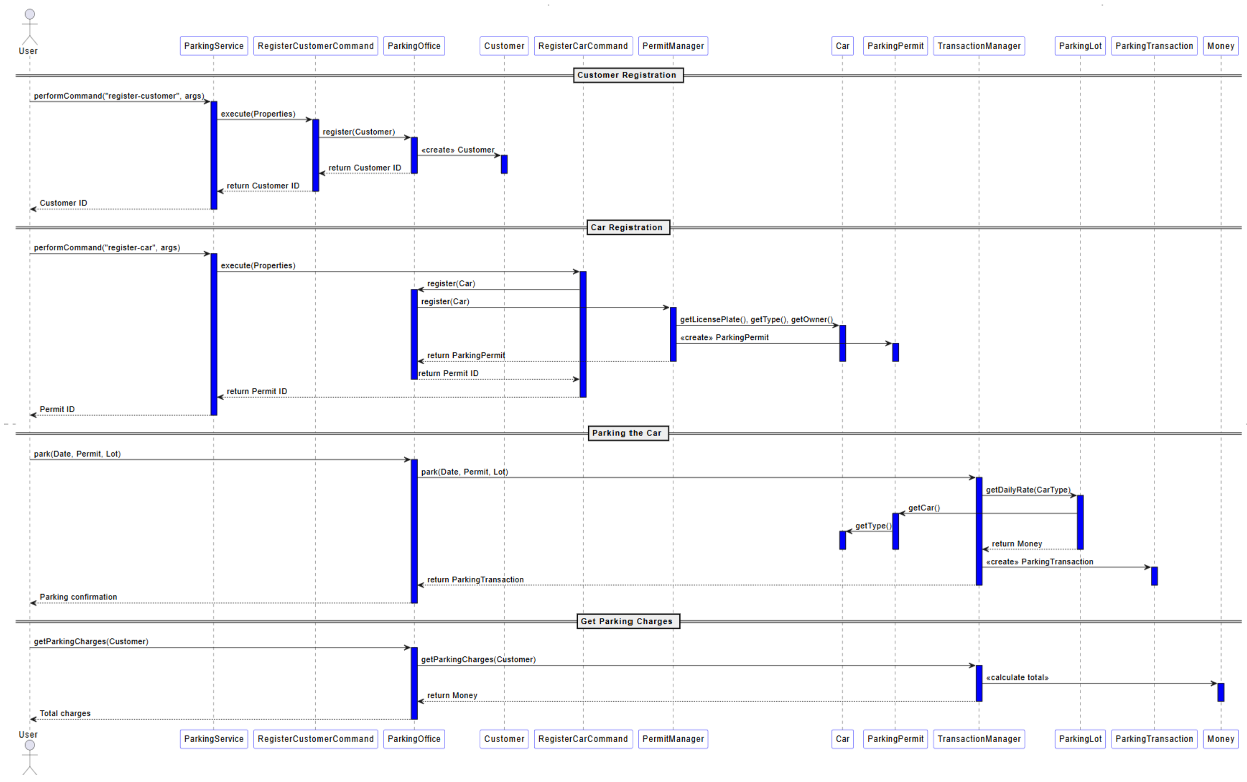


Figure 17: Sequence Diagram

This sequence diagram outlines the main use-case flows in the university parking system, from customer registration to calculating parking charges. It shows how different classes collaborate to complete each task: Customer Registration, Car Registration, Parking the Car and Getting Parking Charges. The key components involved in the sequence diagram include ParkingService, ParkingOffice, RegisterCustomerCommand, RegisterCarCommand, Customer, Car, ParkingPermit, PermitManager, ParkingLot, TransactionManager, Money.

Improvements

One way to enhance the Parking System Application, as highlighted in the introduction, is by further streamlining the registration and parking workflows to improve modularity and decrease direct dependencies between classes. Although the current structure follows a clear sequence and effectively utilizes the Command pattern, introducing dedicated Data Transfer

Objects (DTOs) and refining the responsibilities of service and manager classes could significantly enhance scalability and maintainability. This approach would also make the system easier to test and extend.

Difficulties

One of our initial challenges was designing a system with loosely coupled classes while clearly defining their responsibilities. It was often tricky to determine which class should manage specific tasks. For instance, both the ParkingOffice and the ParkingLot could handle parking transactions, but I ultimately centralized this function within the ParkingOffice to avoid dependencies and redundant functionality.

Implementing the command pattern for car and customer registration through RegisterCustomerCommand and RegisterCarCommand proved complex. Passing Properties objects introduced potential type safety issues and made it harder to trace the flow of operations. Ensuring commands executed in the correct order, with the right side-effects, only added to the challenge.

Error handling also requires a deep focus. I needed to catch issues like missing customer IDs or invalid registrations and provide meaningful feedback, rather than letting errors go unnoticed or resulting in cryptic messages during testing.

Writing unit tests for components like ParkingOffice and ParkingTransaction presented their own obstacles, particularly with external dependencies like the ParkingService class. Mocking these required careful planning to simulate real-world scenarios effectively.

As our system grew, synchronizing interactions between components became increasingly difficult. I had to ensure data flowed smoothly while preserving encapsulation and avoiding circular dependencies—a challenge that became evident when simulating the entire parking process, from registration to calculating charges. Thorough testing was essential to ensure everything worked together seamlessly.

Lessons Learned

- The importance of clearly defining responsibilities for each class early in the design process became evident. At first, I attempted to centralize too much functionality in certain classes, which led to complicated dependencies.
- Testing interactions between commands and data classes was more complex than expected. I learned to create mock objects and stubs for external systems to isolate tests effectively.
- Error handling and validation were crucial for building a system that was not only functional but also resilient and user-friendly. Ensuring meaningful error messages helped us debug and improve the system.

These challenges helped refine the design, leading to a more structured and scalable implementation, but they also revealed the complexities of managing relationships and transactions in an object-oriented system

Testing

JUnit 5.9.3 was used to test the following:

- **Domain Classes (unit tests for object behavior)**
    - <u>CustomerTest</u>: Ensure that customers are registered and retrieved correctly.
        - Test Methods:
            - testCreateFromProperties()

        - testConstructorAndGetters()
        - testSetters()
        - testEqualsAndHashCode()
        - testToString()
        - testNotEquals()
- CarTest: Ensure Car logic is implemented as expected, especially if equality matters.
    - Test Methods:
        - testCreateFromProperties()
        - testConstructorAndGetters()
        - testSetters()
        - testToString()
- ParkingLotTest: Ensure that each parking lot's data and fee structure work as expected.
    - Test Methods:
        - testGetDailyRateCompact()
        - testGetDailyRateSUV()
        - testEqualsAndHashCode()
        - testNotEqualsDifferentId()
- MoneyTest: Validate arithmetic logic and rounding within the Money class.
    - Test Methods:
        - testConstructorAndGetters()

        - testSetters()
        - testEqualsAndHashCode()
        - testToString()
  - AddressTest: Validate the value object storing customer addresses.
    - Test Methods:
      - testNullStreetAddress2()
      - testGetAddressInfo()
  - ParkingPermitTest: Confirm integrity of permit logic and structure.
    - Test Methods:
      - testToStringIncludesImportantInfo()
  - ParkingTransactionTest: Ensure that transactions capture and store all necessary parking details.
    - Test Methods:
      - testConstructorAndGetters()
      - testSetters()
      - testEqualsAndHashCode()
      - testToString()
- **Service and Logic Classes**
  - PermitManagerTest: Validate issuing, storing, and retrieving parking permits.
    - Test Methods:
      - testRegisterPermit()
      - testRegisterCarWithoutOwner()

        - testGetPermitById()
        - testGetInvalidPermitById()
        - testRegisterDuplicateCar()

- TransactionManagerTest**:** Ensure that parking sessions are tracked and fees are calculated accurately.
    - Test Methods:
        - testParkAndCharge()
        - testGetParkingChargesForPermit()
        - testGetParkingChargesForCustomer()
        - testDuplicateTransaction()
        - testTransactionChargesByCustomer()
- ParkingOfficeTest**:** Integration-level test of multiple subsystems (customer, permit, parking, and charges).
    - Test Methods:
        - testRegisterCustomer()
        - testRegisterParkingLot()
        - testRegisterCarAndGeneratePermit()
        - testPark()
        - testGetParkingChargesForPermit()
        - testGetParkingChargesForCustomer()
        - testInvalidCustomerRegistrationForCar()
        - testInvalidParkingLotInPark()

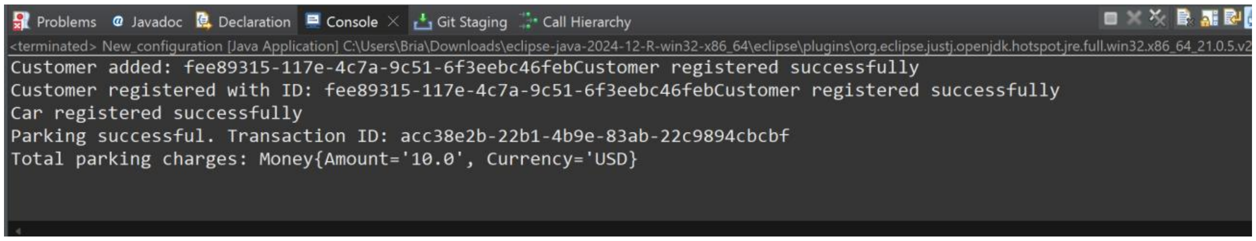
        - testInvalidCarForParking_NoOwner()
  - <u>ParkingServiceTest</u>: Validate the service layer that orchestrates commands and data managers.
    - Test Methods:
      - testPerformCommand_ValidCommand()
      - testPerformCommand_InvalidCommand()
      - testPerformCommand_RegisterCar()
- **Command Classes (Integration with service layer)**
  - <u>RegisterCustomerCommandTest</u>: Verify that the customer registration command triggers service and data layers correctly.
    - Test Methods:
      - testExecute_RegisterCustomer_Success()
      - testExecute_RegisterCustomer_MissingRequiredParameters()
      - testExecute_RegisterCustomer_InvalidCustomerId()
      - testExecute_RegisterCustomer_CustomerAlreadyExists()
  - <u>RegisterCarCommandTest</u>: Verify that car registration (with permit assignment) works as intended via command.
    - Test Methods:
      - testExecute_RegisterCar_Success()
      - testExecute_RegisterCar_CustomerNotFound()

I created fake versions of dependencies to isolate unit tests when necessary, for example, a fake ParkingLot or mocked clock for time-based testing).

The main point for testing the full flow of the application is MainClass in the package ict4315_assignment_1. This class simulates the core use case of the University Parking System: registering a customer and car, creating a parking lot, issuing permit, parking a car, and calculating charges. The purpose of the MainClass execution:

- Validate integration between ParkingOffice, RegisterCarCommand, and data classes (Customer, Car, ParkingLot, ParkingPermit)
- Test the overall parking and charge calculation flow
- Ensure each component is wired correctly and outputs expected results

This acts as a manual end-to-end integration test without needing a UI. If something goes wrong for example, missing fields in Properties, invalid CustomerId, or permits not matching the car, the program prints meaningful error message.



```
<terminated> New_configuration [Java Application] C:\Users\Bria\Downloads\eclipse-java-2024-12-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.5.v2
Customer added: fee89315-117e-4c7a-9c51-6f3eebc46febCustomer registered successfully
Customer registered with ID: fee89315-117e-4c7a-9c51-6f3eebc46febCustomer registered successfully
Car registered successfully
Parking successful. Transaction ID: acc38e2b-22b1-4b9e-83ab-22c9894cbcbf
Total parking charges: Money{Amount='10.0', Currency='USD}
```

Figure 18: MainClass.java output

Conclusion

Engaging in the University Parking System project was a transformative experience that allowed me to effectively apply object-oriented principles in a real-world scenario. This project underscored the critical importance of robust class design, clear separation of concerns, and extensive unit testing. Additionally, I developed a deeper understanding of how helper classes, such as managers, can play a vital role in ensuring organized and efficient core logic. Looking ahead, the potential for this system is immense; by integrating persistent storage, accommodating diverse fee structures (including daily maximums), and creating a user-friendly interface, we can significantly elevate both its functionality and usability, making it truly valuable for users.