Integrated parking management system - Reflection

ICT 4305: Object Oriented Methods and Programming

Hamdi Ali

University of Denver College of Professional Studies

November 14, 2025

Faculty: Nirav Shah, M.S , MBA
Director: Cathie Wilson, M.S.
Dean: Michael J. McGuire, MLS

This final portfolio assignment represents the culmination of my learning throughout the quarter in ICT 4305: Object-Oriented Methods and Programming I. It simulates the operations of a university parking office using a fully functional Parking Management System developed in Java. Object-oriented programming principles such as encapsulation, abstraction, inheritance, and polymorphism are used to track customers, vehicles, parking permits, transactions, and total charges. Based on instructor feedback, I refined the program and gradually integrated specialized manager classes, including PermitManager and TransactionManager, to make the design modular, maintainable, and testable. This leads to a professional-quality application that reflects the iterative and structured development process used in real software engineering projects.

In a data-driven way, the Parking Management System represents how a university or business parking office operates. ParkingOffice serves as the central coordinator and connects all other components. It maintains records of registered customers and parking lots, issues parking permits, and records parking transactions. The specific operational logic is handled by two major supporting classes, PermitManager and TransactionManager. PermitManager creates and manages parking permits, sets expiration dates one year after registration, and links permits to specific vehicles. The TransactionManager records each parking event as a ParkingTransaction, computes charges, and returns the total cost of the permit or customer.

The system's structure is provided by supporting classes. A customer is a registered user. Each vehicle's license plate and type are stored in the car. A parking permit links a car's registration and expiration date. The ParkingLot defines the lot name, hourly rate, and whether scanning or fixed billing is supported. Parking office and customer location details are stored in Address. Currency formatting and arithmetic operations are handled by Money. Billing and transactional information is encapsulated by ParkingCharge and ParkingTransaction. By registering a customer, issuing a permit, parking a car, and printing total charges, the Main class demonstrates how the system works. As a result of this object-oriented structure, the program can accurately mimic the daily operations of a real-life parking office.

The class diagram illustrates the core architecture and relationships among the major classes. A ParkingOffice is at the center, with attributes parkingOfficeName, listOfCustomers, listOfParkingLots, and parkingOfficeAddress. Several methods are exposed to the application: register(Customer) for adding a new customer, register(Car) for issuing a ParkingPermit through PermitManager, park(Date, ParkingPermit, ParkingLot) for recording transactions through TransactionManager, and getParkingCharges(Customer) and getParkingCharges(ParkingPermit) for computing total costs. PermitManager manages the creation and lifecycle of parking permits. It holds a collection of permits and issues new ones using the register method, automatically assigning expiration dates one year after registration. As part of the TransactionManager class, all parking transactions are stored, total charges are calculated, and summary results are provided for customers or individual permits. In the Money class, numerical precision is managed, as well as correct formatting of all charges. In an Address, street information, city information, state information, and ZIP code information are reusable structures. One-to-many relationships are shown in the diagram between Customer and Car, ParkingOffice and ParkingLot, and ParkingOffice and Customer. ParkingTransaction and ParkingPermit are composed because a transaction cannot exist without a corresponding permit. With the overall design, each class is responsible for its own concept, resulting in high cohesion and low coupling.

Sequence diagrams depict how control flows between objects in a typical operation. The process begins when Main calls register(customer) on ParkingOffice. In order to issue a permit, the ParkingOffice class invokes register(car) on PermitManager. The PermitManager returns a ParkingPermit object to the ParkingOffice. Upon parking the car, Main calls park(date, permit, lot) on ParkingOffice, which delegates to TransactionManager to create a ParkingTransaction and record the fee. To calculate the total balance, ParkingOffice calls getParkingCharges(customer) on TransactionManager. The return arrows show how objects such as ParkingPermit, ParkingTransaction, and Money are returned to the caller. This diagram illustrates the delegation of work and confirms ParkingOffice's role as the main controller. The classes are designed to handle only the operations related to their specific roles, maintaining logical flow and clarity.

Unlike the sequence diagram, the interaction diagram emphasizes communication among objects rather than timing. In this diagram, Customer, Car, ParkingOffice, PermitManager, and ParkingPermit collaborate to complete the registration process. ParkingOffice receives a registration request from a Customer. PermitManager is contacted by ParkingOffice by calling register(car). ParkingOffice associates the permit with the customer and car after the PermitManager creates and returns a ParkingPermit. Interfaces are used rather than direct field access to communicate between objects in the diagram. Separating components promotes modularity, reduces coupling, and allows each component to evolve independently. Both static and dynamic aspects of system behavior can be visualized using the interaction diagram and the sequence diagram.

This project strengthened my understanding of object-oriented programming beyond theory. I learned that software design begins with identifying real-world entities and modeling them as classes that have clear responsibilities. A ParkingOffice serves as the system's controller, while PermitManager and TransactionManager encapsulate the business logic. The division of labor makes the program more flexible and easier to maintain. The importance of encapsulation was another key lesson. The classes protect their internal data and expose only the methods that other classes need to use, minimizing the risk of side effects. By representing entities like Car, Customer, and ParkingLot as objects that model real-world concepts while hiding implementation details, abstraction was applied.

Additionally, I gained experience with test-driven development. I created tests for each class using JUnit 5, including ParkingOfficeTest, ParkingLotTest, and MoneyTest. As a result of writing tests before finalizing methods, logic errors were caught early and regressions were prevented. Providing feedback iteratively was crucial to improving the design. Some classes were tightly coupled and difficult to extend early on in the course. With weekly revisions, I refactored them to follow Java conventions, improved naming clarity, and separated logic into manager classes. In the end, the project evolved from a simple procedural design to a well-structured object-oriented system.

In order to achieve this result, I had to use the right tools. The code was developed in IntelliJ IDEA using Java 17. All tests were run and validated using JUnit 5. UML diagrams were created using Lucidchart and PlantUML, which served as visual roadmaps throughout the development process. Version control was provided by GitHub, ensuring that all iterations were backed up and easily accessible. Additionally, these tools simulated the kind of professional workflow used in real-world software engineering environments.

The first thing I would do if I were to start this project over would be to spend more time planning the design in the early stages. Having the UML diagrams at the beginning of the project would have prevented some confusion during implementation and reduced the need for mid-project refactoring. Additionally, I would implement interfaces for manager classes, such as IPermitManager and ITransactionManager, to simplify future integration. There is also room for improvement by adding more modular helper methods within ParkingOffice and TransactionManager so that complex logic can be simplified and readability can be enhanced.

If I had more time, I would add more functionality to the project. Using JDBC or Hibernate, the system could store customer and transaction records persistently. Additionally, I would create a graphical user interface to make the interaction more user-friendly. A dynamic pricing model could be implemented to handle hourly, daily, and weekend rates automatically. It would be helpful to generate detailed monthly and yearly reports to analyze revenue and usage statistics. As a final step, adding authentication and administrative roles would create a complete end-to-end management system.

Completing this project has been one of the most rewarding experiences of my academic career. It demonstrated how programming concepts learned in isolation become part of a comprehensive system. ParkingOffice's integration of PermitManager and TransactionManager illustrates how modular design simplifies complexity. A UML class diagram, sequence diagram, and interaction diagram demonstrate how the different components interact. All unit tests passed, and the final system meets professional standards for readability, maintainability, and accuracy.

This process taught me that object-oriented design is more about organizing thought than writing code. Software engineering requires a clear separation of concerns, iterative testing, and continuous improvement. Parking Management System demonstrates how small, focused classes can model complex real-world processes elegantly. Besides strengthening my understanding of programming, this assignment prepared me for the next course, ICT 4315, where I will incorporate advanced patterns, persistence mechanisms, and possibly web integration into my programming skills. Using both technical competence and conceptual mastery of object-oriented programming, the final product represents a cohesive, tested, and well-documented example of applied software engineering.

References:

GeeksforGeeks. 2024. “Object-Oriented Programming (OOPs) Concepts in Java.” GeeksforGeeks.org. Accessed November 4, 2025. https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/

Oracle. 2024. “The Java™ Tutorials: Object-Oriented Programming Concepts.” Oracle Java Documentation (Java SE 21). Accessed November 4, 2025. https://docs.oracle.com/javase/tutorial/java/concepts/

Oracle. 2024. “Working with Collections in Java.” Oracle Java Documentation. Accessed November 4, 2025. https://docs.oracle.com/javase/tutorial/collections/

JUnit Team. 2025. “JUnit 5 User Guide.” JUnit.org. Accessed November 4, 2025. https://junit.org/junit5/docs/current/user-guide/

Lucidchart. 2025. “What Is UML (Unified Modeling Language)?” Lucidchart Blog. Accessed November 4, 2025. https://www.lucidchart.com/pages/uml

PlantUML. 2025. “Sequence Diagram and Communication Diagram Examples.” PlantUML.com. Accessed November 4, 2025. https://plantuml.com/sequence-diagram

Visual Paradigm. 2025. “Class Diagrams in UML: A Complete Guide.” Visual-Paradigm.com. Accessed November 4, 2025. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

Baeldung. 2025. “Guide to Unit Testing with JUnit 5.” Baeldung.com. Accessed November 4, 2025. https://www.baeldung.com/junit-5