Implement the Parking Charge Calculator for

Master of Science

Information Technology

Bria Wright

University of Denver University College

April 20, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Introduction

The design and implementation of a flexible, modular University Parking System presented a unique opportunity to apply object-oriented programming principles in a real-world context. The system was developed incrementally for a couple of assignments, each week building on the last to add new functionality, improve the class structure, and introduce advanced design patterns. Throughout the project, the requirements evolved and pushed the design of the University Parking System to be loosely coupled, extensible architecture. Special emphasis was placed on abstracting responsibilities into separate layers, allowing for easy expansion or modification of core features, such as parking charge calculations. The ability to introduce new logic, such as weekend discounts or event-day surcharges, without altering the core transaction logic was a central goal of the design. It also outlines the problem-solving approaches used to address these challenges, highlights the importance of design patterns in achieving system flexibility and concludes with the lessons learned from the process. The reflection emphasizes technical growth and an increased understanding of balancing design clarity with complex business requirements.

Design Overview

The University Parking System manages customer registrations, parking permits, vehicle management, and dynamic charge calculations. It needed to support multiple lots with different fee structures, event-based surcharges, and vehicle-specific discounts. I used a modular, object-oriented approach for scalability and flexibility to achieve this. Key user requirements included registering customers and vehicles, assigning multiple cars to one customer, tracking permits, and calculating charges based on rules like entry times and event surcharges. A crucial technical constraint was ensuring the system could adapt to policy changes without major rewrites, which I addressed using the Strategy design pattern for change behavior. Each parking lot can have a distinct charging policy defined by interchangeable pricing strategies, allowing for dynamic charge computation based on factors such as vehicle type, time of day, or special event days (e.g., graduation).

Key design goals included:

- Maintaining loose coupling between classes to support flexibility and extensibility.
- Ensuring encapsulation of business logic within strategy classes to allow easy updates to charge policies.
- Supporting multiple parking lots, each with distinct characteristics and pricing rules.
- Allow multiple cars per customer and manage their respective permits.
- Tracking entry and exit events to produce accurate and auditable transaction records.

Initial assumptions that guided the design included:

- Each car requires a valid permit to enter a parking lot.
- Time-based charges apply, with at least one day of billing for any parking session.
- The system would operate offline in a controlled university setting, where users are pre-registered via an administrative process.
- Graduation events and weekends could affect pricing and must be configurable through strategies, not hard-coded rules.

Technical constraints included limitations on external libraries, adherence to object-oriented design principles, and a requirement to support unit testing using JUnit 5. Additionally, the system needed to be flexible, and it became clear that a more flexible design was required, especially with the introduction of permit-based scanning, customer reports, and varying rates.

Challenges Faced

Throughout the development process, I encountered both expected and unexpected challenges. Initially, I implemented a flat daily rate for all cars. Still, managing logic in a single class became increasingly challenging as requirements grew to include surcharges for early or late entry, graduation events, and compact car discounts. This issue was addressed by introducing a layered strategy pattern, combining behaviors in a chain of responsibility. For example, WeekendFreeStrategy, GraduationSurchargeStrategy, and CompactCarDiscountStrategy could each apply their logic on top of a base strategy without changing the core calculation logic.

Another significant challenge was the **transaction system**, especially in coordinating how charges are calculated, recorded, and associated with specific customer-permit pairs. Introducing a TransactionManager helped centralize this behavior, but getting the system to generate accurate transaction records consistently required meticulous coordination between the ParkingService, Transaction, and ParkingOffice classes. Ensuring that each transaction appropriately included all relevant information—such as entry/exit times, permit details, and calculated fees—led to numerous test revisions and a few rounds of refactoring.

Initially, there were also **misalignments in how object references were managed**, particularly in the relationship between customers, their cars, and the permits tied to those cars. Early system versions allowed permits to be created independently of car registration, resulting in invalid references and broken logic during permit validation. This issue highlighted the importance of ensuring bi-directional consistency in class associations and was ultimately solved by enforcing stricter object creation sequences within the ParkingService layer.

Another challenge was ensuring the ParkingOffice class remained cohesive while supporting many permit and customer lookup methods, transaction handling, and integration with strategy classes. I refactored the design to delegate specific responsibilities to PermitManager and TransactionManager classes to resolve this. This not only reduced complexity but also adhered to the Single Responsibility Principle. I leaned heavily on unit testing to ensure each module worked independently and stayed consistent as new features were introduced. In addition, early mistakes—such as improper use of equals() and hashCode() in collections—caused transaction records to mismatch, which I later resolved through a better understanding of Java's object identity rules.

Despite these hurdles, each challenge ultimately contributed to a more robust and extensible system. The development process emphasized the importance of consistent class responsibilities, interface-driven design, and proactive test coverage—lessons that directly improved the final product.

Problem-Solving Approaches

I adopted several strategies to address these challenges that significantly improved system quality and flexibility. First, I introduced the **Strategy design pattern** to isolate parking charge logic. By decoupling the rate logic from the rest of the system, I could develop independent strategy classes for each scenario—such as WeekendFreeStrategy, GraduationSurchargeStrategy, and CompactCarDiscountStrategy. This allowed me to apply layered behavior while maintaining clean, reusable code.

Additionally, I employed the **Command pattern** while implementing sequence diagram behaviors. This helped translate complex user interactions into modular, testable actions. For example, customer and car registration commands could be passed to the ParkingService, abstracting execution details and making the system more testable and adaptable.
Refactoring was another critical part of my approach. I revised the initial monolithic ParkingOffice design by creating manager classes with focused responsibilities. This shift made the system more maintainable and aligned with the **Single Responsibility Principle**. It also reduced interdependencies between classes, which helped with testing and future extensibility.
To ensure stability and correctness, I incorporated **JUnit 5** for rigorous unit testing and validated each component independently. This helped uncover issues early, particularly with strategy chaining and object identity in collections. I also used assertions and input validation to reduce edge case failures, especially in transaction time validation and permit lookups.

Finally, I remained flexible about initial assumptions. As requirements evolved, I adjusted the data model and flow logic accordingly rather than resisting change. For instance, I revised the belief that a customer would have only one car, enabling multi-car support with permit reuse and dynamic charge tracking.

Key Design Principles

Several design principles guided the development of the system, chief among them being modularity, flexibility, and abstraction:

- Modularity: This was achieved through clearly defined classes and layers. For example, ParkingOffice was designed to coordinate services but not handle the specifics of permit or transaction logic directly. Those responsibilities were passed to PermitManager and TransactionManager, which could be tested independently or reused in other contexts.

- Flexibility: was a constant concern due to evolving requirements. The charge strategy system was implemented with the Strategy and Decorator patterns in mind, which made it easy to plug in new behaviors without changing core logic. For example, adding a surcharge for graduation day was as simple as writing a new strategy and chaining it with existing ones.
- Abstraction: played a role in isolating change. Interfaces like ParkingChargeStrategy and Command allowed the implementation to change without affecting clients. For example, the command pattern was later introduced to help translate sequence diagrams into behavior. Instead of hard-coding method calls, commands could now be created and passed to a service, simplifying testing and reducing tight coupling.

Lessons Learned

This project was an eye-opener in many ways. Technically, I learned the importance of handling Java collections correctly—especially the need to override equals() and hashCode() in domain classes like Customer and Car. Without doing so, identity issues would prevent proper list filtering or set membership. One of the most essential technical lessons was the effective use of design patterns, particularly the Strategy and Decorator patterns. While I had previously studied these patterns in theory, this project allowed me to apply them in practice. The Strategy pattern allowed the system to support multiple parking charge calculations without hardcoding logic into a single class. The Decorator pattern proved invaluable when I introduced additional logic—like the Graduation Day surcharge, compact car discounts, or early entry fees. It let me stack these rules cleanly, clearly separating concerns.

Another key area of growth was exception handling. Initially, my code often assumed everything would go well, but through testing, I learned the value of guarding against null values, invalid IDs, and expired permits. After encountering several runtime issues during testing, I revised the application to include more thorough validation and exception handling. This improved the system's reliability and helped prevent cascading failures when dealing with user errors or incomplete data.

From a design standpoint, I learned how initial assumptions can shift quickly. For example, I initially assumed the pricing model would be static, with only weekday rates to consider. However, as the requirements evolved to include event surcharges and time-based adjustments, I had to pivot toward a more flexible and extensible pricing strategy. This taught me the importance of designing with change in mind, even when requirements seem fixed initially. On a broader level, this project reinforced the importance of iterative development and consistent refactoring. As new features were added each week—from multi-car permits to transaction tracking—I revisited earlier code to improve readability, remove duplication, and align with newly introduced structures. Maintaining clean, modular code from the beginning made these changes far less painful than they might have been in a more tightly coupled design. The biggest lesson was how crucial refactoring is in real-world development. The initial ParkingOffice class ballooned in responsibility until I introduced dedicated managers. Likewise, charge calculation strategies were initially embedded directly in if-statements until I separated them into modular classes. These changes improved clarity, reusability, and confidence in the system's correctness.

This project deepened my understanding of good software design practices, particularly in object-oriented programming. It also highlighted the importance of planning for change, validating assumptions, and building software with manageable, testable components. I will carry these lessons forward into future development work.

Conclusion

This university parking system project explored real-world software design in-depth, from initial planning and user-centered objectives to implementing scalable, flexible features. The core design successfully met its primary goals: providing a structured, modular system for customer registration, car management, permit handling, and calculating parking charges under varying conditions.

Despite the challenges—such as evolving requirements, integrating multiple strategies into a cohesive charge calculation process, and managing edge cases—the project remained on track due to careful design, constant testing, and iterative improvement. Implementing loosely coupled classes and defining clear responsibilities across components helped resolve early integration difficulties while adopting extensible strategies made adding features like event surcharges and discount policies easier.

Completing this assignment gave me a much deeper appreciation for the balance between planning and adaptability. I've learned to prioritize clarity and modularity in my code, and I now better understand the importance of test-driven development and documentation in maintaining complex systems over time. These lessons have reshaped my development mindset, and I feel more confident tackling future projects, especially those with evolving goals or domain-specific requirements.

REFERENCES

GeeksForGeeks. 2017. “Inheritance in Java - GeeksforGeeks.” GeeksforGeeks. March 23, 2017. https://www.geeksforgeeks.org/inheritance-in-java/.

Kaminski, Ted. 2018. “What’s Wrong with Inheritance?” Tedinski.com. February 13, 2018. https://www.tedinski.com/2018/02/13/inheritance-modularity.html?.

Nagendra, Saajan. 2020. “The Liskov Substitution Principle Explained.” Reflectoring.io. July 5, 2020. https://reflectoring.io/lsp-explained/.

“Structural Design Patterns.” 2023. GeeksforGeeks. December 1, 2023. https://www.geeksforgeeks.org/structural-design-patterns/.

Team, Great Learning. 2021. “What Is Inheritance in Java and Types of Inheritance in Java.” GreatLearning Blog: Free Resources What Matters to Shape Your Career! May 27, 2021. https://www.mygreatlearning.com/blog/inheritance-in-java/.