Implementing a Creational Pattern: Factory for

Master of Science

Information Technology

Bria Wright

University of Denver University College

April 27, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Introduction

This project focused on extending and improving the design of the University Parking System. Using the Strategy and Factory design patterns, where the strategies introduced a flexible charge calculation mechanism. The goal was to create a system where parking lots could dynamically apply different charging strategies without changing the core system logic. I focused on making the refinements flexible, easy to test, and extensible for future changes. In addition, I created the ParkingChargeStrategy, integrating it into the ParkingLot and TransactionManager to manage parking transactions using the correct strategy. Also, implementing unit tests to ensure everything worked correctly. Then, I designed a ParkingChargeStrategyFactory to select the correct strategy based on parking lot characteristics. Different lots could have unique charging behaviors without requiring significant code changes. Throughout the assignment, I prioritized clean design, testability, and extensibility while encountering and solving several design challenges.

Design Overview

The updated system centers on several key enhancements:

- ParkingChargeStrategy Interface: This interface establishes standard methods for calculating charges (calculateCharge) and determining if the fee is based on an hourly rate (isHourlyBased) (W3Schools 2025).
- Concrete Strategy Classes: Implementations such as HourlyRateStrategy, FlatDailyRateStrategy, WeekendFreeStrategy, GraduationSurchargeStrategy, and FlatDailyRateWithCompactDiscountStrategy embody various business rules governing parking fees (GeeksforGeeks 2023).

- DefaultParkingChargeStrategyFactory: This component centralizes the logic for selecting the appropriate charge strategy based on the parking lot's characteristics or name (Great Learning 2024).
- TransactionManager: This has been refactored to dynamically request a charge strategy from the factory rather than relying on a hardcoded approach (Baeldung et al. 2024).
- ParkingOffice: Refactored to accept a ParkingChargeStrategyFactory through constructor injection, facilitating better dependency management (Gupta 2024).

The system now accommodates flexible and scalable parking charge rules without modifying the underlying code.

Challenges Faced

Throughout the evolution of the ParkingChargeStrategyFactory, I encountered several challenges that mirrored those from the previous week. My primary task involved aligning parking lots with their corresponding strategies and devising methods to associate these lots with various pricing rules that adhered to the strategies. Additionally, I faced difficulties with the requirements of specific strategy constructors, such as WeekendFreeStrategy and GraduationSurchargeStrategy, which necessitated additional parameters (Simplilearn 2024). Ensuring adequate testing coverage was also a challenge, as I needed to calculate the pricing accurately and initialize the ParkingChargeStrategyFactory effectively.

Regarding linking parking lots to strategies, the initial system did not effectively associate lots with distinct pricing rules. Determining how to match a lot with a strategy emerged as the first significant challenge. I contemplated adding a type of attribute to ParkingLot. Still, I opted

for simplicity by using keywords in the lot's name (such as "hourly" or "weekend") to identify the appropriate strategy, employing 'if' statements to cover the various techniques.

Specific strategies, including WeekendFreeStrategy and GraduationSurchargeStrategy, required additional parameters, such as dates, which were not readily available in the current factory setup at the time of strategy selection. Managing these requirements without complicating the design further presented another obstacle.

Finally, ensuring thorough testing proved to be a significant challenge. Initially, I developed basic tests for one or two strategies, but I soon recognized the need for comprehensive tests to cover all possible lot types and error conditions.

Problem-Solving Approaches

- String Matching for Efficient Lot Identification: I implemented a straightforward yet effective string-matching technique, leveraging each parking lot's unique names to swiftly associate them with their respective strategies. Although this method may not represent the best option of robustness, it proved instrumental in achieving the assignment's goals without necessitating extensive structural modifications to the parking lot framework (W3Schools 2025).
- Robust Handling of Null Parameters: To deftly manage strategies that required supplementary inputs, I temporarily introduced null values into the constructor where appropriate. While this solution may lack elegance, it safeguards against potential runtime crashes. For a more robust production environment, I would advocate developing a builder or factory method to provide all necessary configuration data, enhancing overall system reliability (Martin 2024).

- Incremental Test Development: I adopted a strategic approach to testing that emphasized incremental development. I started with fundamental tests and systematically expanded them to assess each strategy type in isolation. This method guaranteed the system's ability to manage unknown lots effectively and facilitated the early identification of missing conditions and logical inconsistencies, ensuring a more thorough solution (Martin 2024).

Key Design Principles

Several object-oriented design principles significantly informed my implementation, enhancing the clarity and efficiency of the code:

- Open/Closed Principle: This principle ensures that software entities (like classes and modules) can be extended without modifying their existing code. My implementation exemplified this by introducing a WeekendFreeStrategy for calculating parking charges. To incorporate this new strategy, I updated the ParkingChargeStrategyFactory to include the new strategy type, which allows the system to recognize and utilize it. This approach kept the core functionalities of the TransactionManager and ParkingOffice intact, thus preventing potential bugs that could arise from modifying established classes (GeeksforGeeks 2020).
- Single Responsibility Principle: Adhering to this principle, I structured each class to have a singular, well-defined responsibility. For example, the ParkingChargeStrategyFactory is tasked solely with selecting the appropriate charging strategy based on the context, while the various charge strategies themselves are responsible for implementing the specific calculations. The ParkingOffice orchestrates the interaction between these

components, ensuring a clear division of roles that promotes easier maintenance and understanding of the codebase (Gupta 2024).

- Dependency Injection: By employing dependency injection, the ParkingOffice class receives its instance of ParkingChargeStrategyFactory through its constructor rather than creating it internally. This design choice fosters greater flexibility in substituting different strategies and simplifies unit testing. For instance, I can easily mock the factory during testing to return predefined strategies, allowing for isolated testing of the ParkingOffice and ParkingLot without setting up the entire environment (Baeldung et al. 2024).
- Polymorphism: Implementing the Strategy Pattern allowed the seamless integration of diverse charge calculation methods into the broader system. By defining a standard interface for all charge strategies, different implementations can be swapped in and out without affecting the client code that interacts with them. This minimizes the need for modifications in multiple places, making the system more adaptable to future changes or enhancements (Sandamini 2023).

These principles contributed to clean and organized architecture and allowed for an extensible and easily testable codebase throughout the development process, ultimately improving maintainability and reducing technical debt.

Lessons Learned

- Early Design Decisions Matter: In the initial phases of the project, I opted for a simple keyword-matching system for lot names, which allowed for rapid implementation and quick turnaround. However, this decision led to significant long-term issues, as the system became increasingly fragile and prone to errors over time. For future iterations, I

would advocate for implementing a more robust and type-safe approach, such as using enums or well-structured configuration classes to define lot types. This shifts the focus from string manipulation to clear type definitions, ultimately enhancing the system's resilience and reducing the likelihood of misconfiguration (Great Learning 2024).

- Testing is Critical: The importance of rigorous testing cannot be overstated. During the development of the factory pattern, I implemented unit tests that served as a safety net, uncovering subtle logic errors early in the development process. This proactive approach not only caught bugs before they escalated into significant issues but also bolstered the system's overall reliability. The confidence gained from comprehensive testing practices enabled me to make iterative improvements with the assurance that existing functionality remained intact (Martin 2024).
- Factory and Strategy Patterns are Powerful: Incorporating Factory and Strategy design patterns profoundly improved the project's architecture. By decoupling object creation from business logic, I could create a more modular and flexible codebase that was easier to maintain and extend. For instance, separating the instantiation logic from the decision-making processes made it simpler to swap out implementations and introduce new features without disrupting existing functionality. This modularity allowed for greater adaptability in responding to changing project requirements (Baeldung et al. 2024).
- Trade-offs are Necessary: One decision involved passing null values into constructors as a temporary workaround to handle optional parameters. While this approach met immediate needs and allowed for quick development, it illuminated the challenges of

clean parameter handling in more complex, real-world applications. This experience underscored the necessity of prioritizing robust design practices, such as utilizing optional parameters or well-defined default values, to enhance code clarity and maintainability in the long run (Martin 2024).

Conclusion

This assignment offered helpful hands-on experience in applying Strategy and Factory design patterns to a real-world scenario. By effectively distinguishing between different concerns and prioritizing design for extensibility over mere modification, I developed a sophisticated parking system capable of accommodating a variety of pricing strategies with remarkable ease. Throughout the development process, I encountered challenges related to lot identification and the instantiation of strategies. However, through diligent problem-solving and thorough testing, I was able to navigate these obstacles and achieve a successful outcome.

I take great pride in the system's design, which reflects a deep understanding of object-oriented principles. This experience has significantly boosted my confidence in applying these concepts to future projects, reinforcing my commitment to creating robust and adaptable software solutions.

References

Baeldung, et al. 2024. "Factory Design Pattern in Java." *Baeldung.com*. https://www.baeldung.com/java-factory-pattern.

GeeksforGeeks. 2020. “Factory Method Design Pattern in Java.” GeeksforGeeks. April 28, 2020. https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/#what-is-the-factory-method-design-pattern-in-java.

———. 2023. "Strategy Design Pattern - GFG." *GeeksforGeeks.org*. https://www.geeksforgeeks.org/strategy-pattern/.

Great Learning. 2024. "Design Patterns in Java: Strategy and Factory Explained." *GreatLearning.com*. https://www.mygreatlearning.com/academy/learn/java-design-patterns.

Gupta, Ramesh. 2024. "Mastering Object-Oriented Design with Java." *Techie Blog*.

Martin, Robert C. 2024. *Clean Code: A Handbook of Agile Software Craftsmanship*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Sandamini. 2023. "Introduction to Java Inheritance and Interfaces." *Java Tutorials Online*. https://www.javatutorials.com/inheritance-interfaces.

Simplilearn. 2024. "Factory and Strategy Pattern in Java." *Simplilearn.com*. https://www.simplilearn.com/tutorials/java-tutorial/factory-and-strategy-pattern-in-java.

W3Schools. 2025. "Java Design Patterns." *W3Schools.com*. https://www.w3schools.com/java/java_design_patterns.asp.