Decorating Our Parking Charges for

Master of Science

Information Technology

Bria Wright

University of Denver University College

May 11, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Introduction

The University Parking System is a parking management system simulation where vehicles are registered, permitted, and charged based on various contextual factors. Initially, I implemented the parking fee algorithm using the Strategy design pattern, allowing me to swap out charging strategies based on parking lot types. However, as the number of modifiers (like compact car discounts, overnight discounts, and event day surcharges) increased, the Strategy pattern began to show limitations. I transitioned from Strategy to Decorator to modularize the charging logic better, how the Decorator was implemented, challenges I encountered, and why I ultimately preferred Decorator for this use case.

Design Overview

The original implementation effectively employed the Strategy pattern, featuring a flat or variable rate strategy encapsulated in a single method that took parameters like the permit, entry, and exit times. Different parking lots confidently utilized distinct strategies (such as FlatDailyRateStrategy and HourlyRateStrategy) to compute charges by assessing applicable modifiers. However, as more conditional checks were added, the strategy class became bloated, complicating testing and making it challenging to extend without altering existing code, ultimately violating the Open/Closed Principle (Martin 2008).

To tackle this issue head-on, I implemented the Decorator pattern, which empowers it to wrap a basic calculator with additional decorators designed to apply specific rules independently. This approach aligns with the structural flexibility described in *Design Patterns: Elements of Reusable Object-Oriented Software*, where decorators allow adding functionality to objects at runtime without modifying their class (Gamma et al. 1994). For instance, decorators

like CompactCarDiscountDecorator, OvernightDiscountDecorator, and GameDaySurchargeDecorator each modify the base charge according to their isolated logic. This new design includes:

- An abstract base class, ParkingChargeCalculator, with the getParkingCharge() method.
- A concrete base class, FlatRateCalculator, efficiently computes the basic charge.
- An abstract decorator, ParkingChargeCalculatorDecorator, which holds a reference to another ParkingChargeCalculator.
- Concrete decorators, such as CompactCarDiscountDecorator, implement specific pricing rules and return modified results.

By introducing this layered architecture, I improved code clarity and modularity and enabled easier testing and extension. Each component adheres to the Single Responsibility and Open/Closed principles (Martin 2008). It also reflects the compositional reuse strategy Gamma et al. (1994) advocated, wherein behavior is composed dynamically rather than embedded in rigid inheritance hierarchies.

Design Explanation

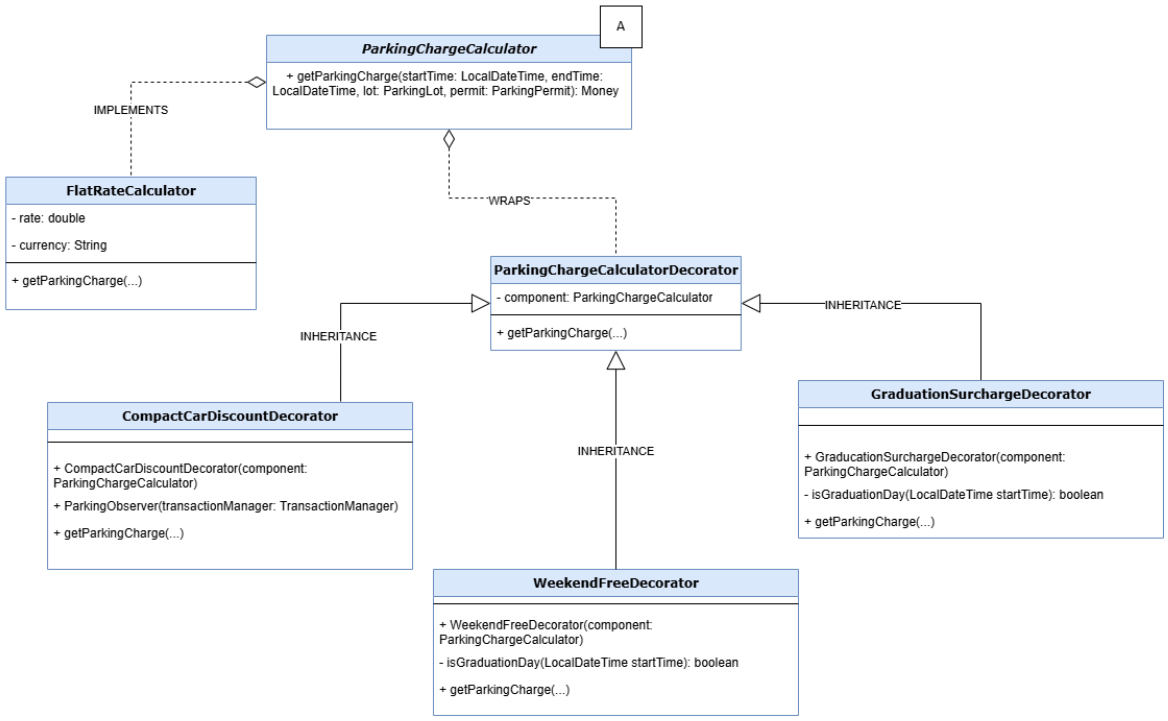

Figure 1: UML Class Diagram – Decorators

The class diagram (see attached) visually represents how decorators dynamically wrap the FlatRateCalculator to apply discounts or surcharges. The key relationships are:

- ParkingChargeCalculator is the abstract base type for all calculators.
- FlatRateCalculator provides the baseline rate implementation.
- Decorators such as CompactCarDiscountDecorator extend ParkingChargeCalculator and hold a reference to another calculator, enabling recursive delegation of charge calculations.

This composition-based design supports modular extension without the risks of tight coupling, aligning with the refactorability and maintainability goals emphasized in *Clean Code* (Martin 2008).

Comparison of Strategy vs. Decorator

In our case, the Decorator pattern offers superior flexibility and adheres more closely to SOLID principles, especially Open/Closed and Single Responsibility (Martin, 2002).

Table 1. Strategy Pattern vs. Decorator Pattern

Feature	Strategy Pattern	Decorator Pattern
Flexibility	Moderate – one algorithm per strategy	High – modular, combinable factors
Extensibility	Poor – adding factors requires editing code	Excellent – add new decorators independently
Maintainability	Declines as rules increase	Remains high due to modularity
Testing Simplicity	Simple for few rules	Requires unit tests per decorator
Design Complexity	Simpler to implement	More complex but structured

Challenges and Solutions

Decorator Chaining Order

One of the initial and most significant challenges I faced in transitioning to the Decorator pattern was ensuring the integrity of the calculation chain while maintaining modularity. Unlike the Strategy implementation, where all business logic was centralized in a single method, making it straightforward but less flexible, the shift to the Decorator pattern required a more sophisticated approach. Each condition, such as compact car discounts or weekend rates, was clearly defined within its class, effectively wrapping the preceding logic.

Every ParkingChargeDecorator subclass was designed to invoke the wrapped component's getParkingCharge(...) method, apply its specific business rule, and return a modified result. I understood the importance of executing this correctly, as any mistake could lead to overwriting previous calculations or applying them multiple times. This necessitated careful method delegation and internal logic checks to ensure that decorators were chained efficiently without duplicating or omitting essential computation steps.

Furthermore, the order in which decorators were applied was critical, for instance, using a fixed-rate surcharge after a percentage discount results in a different total compared to applying it beforehand. To manage this complexity, I established a standardized sequence through a centralized ParkingChargeCalculatorFactory, which constructs the appropriate chain of decorators based on lot type, time, and vehicle conditions. I refined this factory pattern to assemble decorators in a predictable and maintainable manner, ensuring the consistent application of all relevant pricing factors. This approach enhances clarity and guarantees reliability across various pricing scenarios.

Testing Combinations

Testing has significantly evolved within this framework. In the strategy-based approach, a single test could initiate a comprehensive charge calculation for multiple conditions. Each rule was encapsulated in its class with the Decorator pattern, necessitating unit tests for individual decorators and integration tests for the composed chains. Each decorator was tested in isolation to ensure that it accurately modified the output based on the expected inputs.

For integration testing, I developed various combinations (e.g., compact + weekend, compact + surcharge, all three together, etc.) and compared the final charge against known

values. Using the weekend decorator initially resulted in a zero charge for every event, even on Graduation Day. To address this issue, I implemented a boolean variable called isGraduationDay. This allows me to set Graduation Day and other future events to true, ensuring that the lot charges are calculated correctly. This allowed me to verify that the decorators interacted correctly and that the cumulative effects matched our expectations. Achieving this required robust test coverage and careful composition of test cases.

Access to Required Data

A key design consideration was ensuring access to contextual data—such as permit details, vehicle type, and parking duration—without tightly coupling decorators to the core system. Each decorator required sufficient information to make informed, context-aware decisions, like determining compact discount eligibility. By avoiding global data injection and keeping decorators stateless, I passed contextual objects—ParkingPermit, ParkingLot, and timestamps—as method parameters.

This approach preserved low coupling and ensured that each decorator remained stateless and reusable, in line with SOLID principles, particularly Single Responsibility and Open/Closed (Martin, 2002). Each decorator modified the charge based solely on the provided data, thereby enhancing the system's maintainability and testability.

Conclusion

Implementing and comparing the Strategy and Decorator design patterns highlighted that the Decorator pattern provides a more scalable and maintainable solution for calculating parking charges. As pricing factors increased, the Strategy pattern became inflexible, violating key SOLID principles like Open/Closed and Single Responsibility. The Decorator pattern,

however, allowed for independent composition of pricing rules, facilitating clean extensions and improving maintainability. While it introduced some complexity in chaining and testing, these challenges were managed effectively. Ultimately, the Decorator pattern proved to be the most adaptable and future-proof choice for the University Parking System.

References

Martin, Robert C. 2008. “Clean Code: A Handbook of Agile Software Craftsmanship [Book].” Www.oreilly.com. August 2008. https://learning.oreilly.com/library/view/clean-code-a/9780136083238/.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. *GitHub*. Pearson Education. https://github.com/TushaarGVS/Design-Patterns-Mentorship/tree/master.

Refactoring Guru. 2014. “Observer.” Refactoring.guru. 2014. https://refactoring.guru/design-patterns/observer.