Decorating Our Parking Charges

for

Master of Science

Information Technology

Peter Fedor

University of Denver College of Professional Studies

May 11, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

For this assignment, I was tasked with implementing a new Decorator design pattern into my existing Parking System Application. With the Decorator pattern, I can now mix and match different pricing rules for each lot or vehicle without rewriting large chunks of code. It allows the system to grow in a clean and organized way. I can apply a discount for compact cars, add a weekend surcharge, or even handle overnight pricing just by stacking decorators—no need to create a new class for every possible combination of rules.

Design

For the design, I started by creating an abstract class called ParkingChargeCalculator. This class acts as the foundation for all pricing logic and defines a method called getParkingCharge, which takes in entry and exit times, the parking lot, and a permit. The idea here was to centralize charge logic so everything builds off this common interface. I then wrote a simple base implementation called FlatRateCalculator. This just charges a flat fee, regardless of how long the vehicle is parked. From there, I added the decorators. I created an abstract class, ParkingChargeCalculatorDecorator, which wraps another ParkingChargeCalculator. This wrapper allows you to build up logic in layers. For example, the CompactCarDiscountDecorator checks the car type from the permit, and if it's a compact car, it reduces the fee by 20%. The decorators are designed to be added dynamically, depending on the lot or vehicle. This was a big shift from the earlier Strategy-based design where I had to define a specific charge strategy for each case. Now, I can combine different decorators to get the price I want.

Challenges

I found this assignment, like the others, to be a bit tricky. I've never used the Decorator

pattern in Java before, so just understanding the structure took some time. At first, I thought decorators were just subclasses that added extra behavior. But after digging deeper, I realized they're more like wrappers—you build them around a base object, and they can call the base method and add their own logic on top. The real challenge came in integrating this with my existing code. My Parking System was already structured around the Strategy pattern, so I had to carefully replace that logic without breaking the core flow. It took a few tries to get the decorators working properly with the TransactionManager and ParkingLot classes. Another challenge was testing. With the Strategy pattern, each charge type was isolated and easy to test. But now, decorators can be stacked in different ways, so I had to test multiple combinations to make sure the logic was being applied correctly. For example, a compact car using a lot with a flat rate and a weekend surcharge might need two or three decorators applied in sequence. Getting those unit tests to reflect real-world use cases took some thought. Overall, though, I found the process rewarding. It forced me to think more deeply about object composition and dynamic behavior, rather than just inheritance and fixed structures.

Conclusion

Looking back, I can clearly see the difference between using the Strategy pattern and the Decorator pattern for this project. The Strategy pattern was helpful when I just needed to plug in a single pricing model like hourly or daily. But it quickly became rigid as soon as I wanted to apply multiple pricing rules at once—like a flat fee and a discount for compact cars. The Decorator pattern solves this problem neatly. It lets me combine behaviors on the fly and keeps each rule self-contained. If I want to add an event surcharge or change how discounts are applied, I can just write a new decorator and wrap it around the existing ones. There's no need to touch the core calculation logic. That said, the Strategy pattern was simpler to understand and implement. For smaller systems

or where pricing rules don’t change often, Strategy might still be the better choice. But for this project, which needs to support multiple vehicle types, flexible pricing, and potential future growth, the Decorator pattern makes more sense. In the end, I think the Decorator approach is the better fit for this problem. It offers the extensibility I need without locking me into coupled class structures.

References

"Decorator Design Pattern in Java with Example." 2021. GeeksforGeeks. March 8, 2021. https://www.geeksforgeeks.org/decorator-design-pattern-in-java-with-example/.

"Decorator." n.d. Refactoring.guru. https://refactoring.guru/design-patterns/decorator.