Week 6 Assignment: Using the Observer Pattern to Assess Parking Charges for

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

May 11, 2025

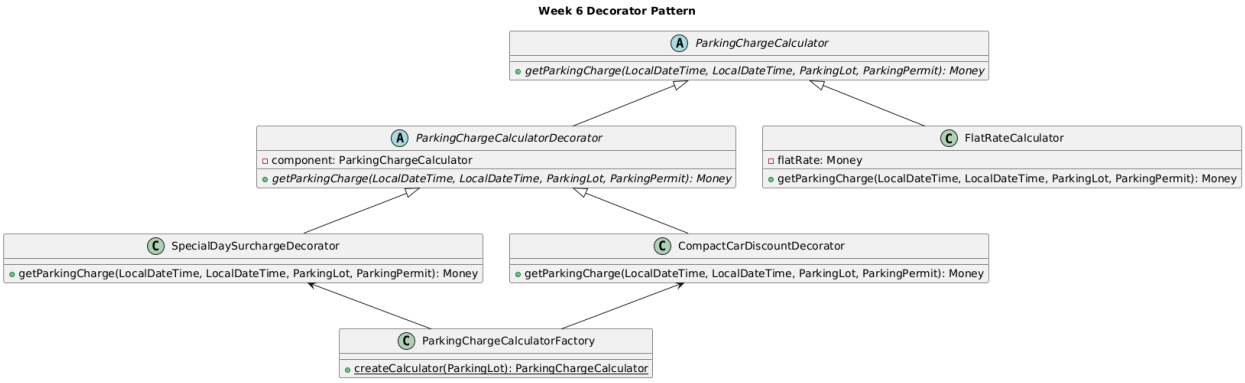
Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

This paper discusses the implementation of the “decorator pattern”, which uses decorators, also known as wrappers. The decorator pattern is a structural design pattern that adds functionality to objects by placing them inside special “wrapper” classes that contain these additional behaviors (Refactoring Guru, 2025).

To better understand how this applies to the parking lot project, it helps to look at the UML diagram. This diagram is also attached in the assignment submission where it can be viewed at full size for better legibility.



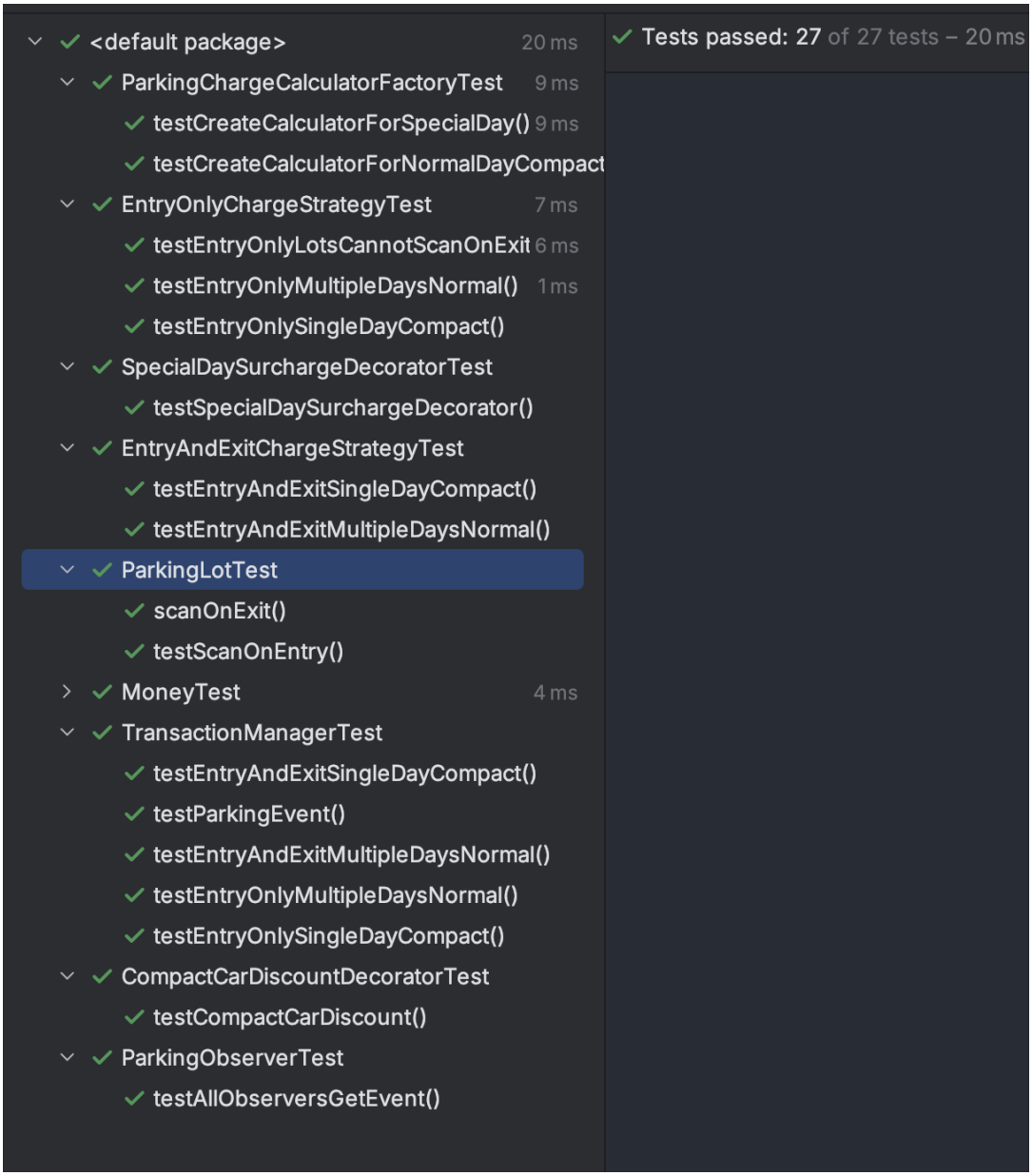
We start with an abstract Parking Charge Calculator class that declares a method “get parking charge” and takes the different pieces of information required to compute a charge. On one hand, the Flat Rate Calculator concrete class simply extends the abstract class and implements the “get parking charge” method using the parking lot’s flat rate. On the other hand, the Parking Charge Calculator Decorator is an abstract class that extends the Parking Charge Calculator, adding a reference to a “component”, which “declares the common interface for both wrappers and wrapped objects” (Refactoring Guru, 2025). From there, two concrete decorators extend that abstract class and modify the behavior.

The Special Day Surcharge Decorator adds functionality for special days like graduation, applying a different strategy (from previous assignments). The Compact Car Discount Decorator similarly adds functionality for giving discounts to compact cars, applying another strategy.

One tricky issue I encountered was that my previous implementation expected a vehicle type (compact or normal) in order to return the rate for each lot. This was useful at the time because the rate changes based on the type of vehicle. However, this pattern doesn't work the same way. At one point in the development of this week's project, the compact vehicle discount was being applied twice. To solve this, I created an override of the getter for the rate that does not require a car type, leaving it up to the caller to figure out what discounts should be applied. In this case, the decorator modified the behavior and applied a discount, solving the problem.

Overall, the decorator pattern is a good way to modify and extend the behavior of methods to handle more specific use cases, like special events or compact cars. The original method simply returns a flat rate, while the decorators add extra functionality. More decorators can be added on to handle other use cases.

The code compiles and all tests pass:



References

Refactoring Guru. 2025. "Decorator." Accessed May 11, 2025. https://refactoring.guru/design-patterns/decorator.