Using the Observer Pattern to Assess Parking Charges for

Master of Science

Information Technology

Bria Wright

University of Denver University College

May 4, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Introduction

Modern parking systems require real-time responsiveness and a modular design to effectively manage dynamic vehicle activity. Implementing established design patterns is crucial for developing scalable and maintainable software. This assignment focused on applying the Observer design pattern within a University Parking System to facilitate event-driven communication between key components. The goal was to simulate a parking lot that can notify interested observers, such as transaction managers, whenever vehicles enter or exit. By decoupling event generation from event handling, I enhanced the system's flexibility, reduce complexity, and pave the way for future extensions, such as alert systems or analytics modules.

Design Overview

This assignment focused on implementing the Observer design pattern within the University Parking System to enhance both the modularity and responsiveness of the application. The Observer Design Pattern is a behavioral pattern that establishes a "one-to-many dependency between objects." When the state of one object (the subject) changes, all its dependents (observers) are promptly notified and updated automatically (GeeksForGeeks 2016a). This pattern effectively simulates real-world scenarios in which parking events—such as vehicle entries and exits—trigger notifications to observers monitoring parking activities, thereby prompting the appropriate system responses.

The implementation involved three core classes: ParkingLot, ParkingObserver, and TransactionManager. The ParkingLot class serves as the representation of the parking facility and functions as the Subject (GeeksForGeeks 2016a). The ParkingObserver monitors the state of the parking lot, while the TransactionManager is responsible for logging new parking transactions. This design promotes a loose coupling between event producers (the parking lots) and consumers (transaction managers and other observers), ultimately enhancing the system's maintainability and extensibility.

Design Explanation

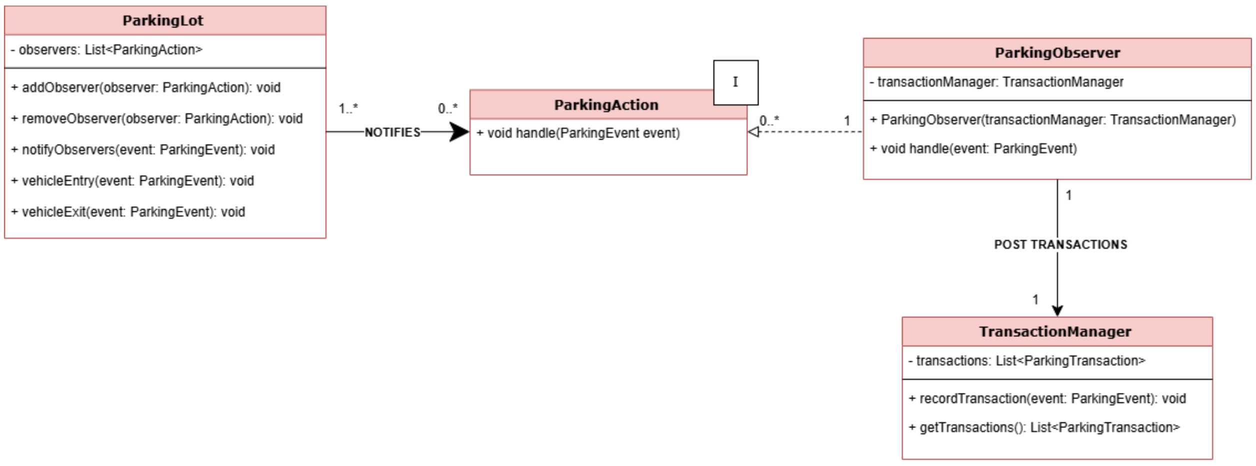
ParkingLot
- observers: List<ParkingAction>
+ addObserver(observer: ParkingAction): void
+ removeObserver(observer: ParkingAction): void
+ notifyObservers(event: ParkingEvent): void
+ vehicleEntry(event: ParkingEvent): void
+ vehicleExit(event: ParkingEvent): void
1..*
NOTIFIES
0..*
ParkingAction
I
+ void handle(ParkingEvent event)
0..*
1
ParkingObserver
- transactionManager: TransactionManager
+ ParkingObserver(transactionManager: TransactionManager)
+ void handle(event: ParkingEvent)
1
POST TRANSACTIONS
1
TransactionManager
- transactions: List<ParkingTransaction>
+ recordTransaction(event: ParkingEvent): void
+ getTransactions(): List<ParkingTransaction>

Figure 1: UML Class Diagram - Observers

I created a UML class diagram focusing on the new attributes and methods contributing to the ParkingObserver class. Here are the relationships between the classes:

- ParkingLot (Subject) (Refactoring Guru 2014):
    - This class manages parking events such as vehicle entry and exit. It holds a list of ParkingAction observers.
    - When an event (e.g., entry or exit) occurs, it notifies all attached observers via their notifyObservers() methods.
- ParkingAction (Observer Interface):
    - This interface defines the method void update(ParkingEvent event) implemented by any observer interested in receiving parking event notifications.
- ParkingObserver (Concrete Observer):

    - Implements ParkingAction.
    - Defines the behavior upon receiving a ParkingEvent. In this case, it translates the event into a ParkingTransaction and delegates it to the TransactionManager.
- TransactionManager:
    - Responsible for creating and storing ParkingTransaction objects based on incoming ParkingEvents.
    - While it is not an observer, ParkingObserver uses it to finalize the event lifecycle by recording the transaction.

This architecture ensures that ParkingLot remains neutral regarding how parking events are managed beyond notifications, while the TransactionManager is separated from the process of generating events.

Pattern Analysis

This implementation serves as a definitive example of the Observer pattern as outlined by Gamma et al. (1994) and consistently emphasized in leading software design literature. The fundamental principle states, "define a one-to-many dependency between objects so that when one object changes its state, all its dependents are automatically notified and updated."

In this scenario:

- Subject: The ParkingLot robustly maintains a list of observers (ParkingAction) and offers methods for adding and removing these observers. It efficiently calls the update() method on all observers when the notifyObservers(ParkingEvent) function is triggered (GeeksForGeeks 2016a).

- Observers: These are dynamic implementations of ParkingAction that respond to updates in real-time without requiring any modifications to the ParkingLot code. They adeptly translate updates into essential business logic, such as saving transactions (GeeksForGeeks 2016a).

The benefits of this pattern are significant:

- Loose Coupling: The ParkingLot operates independently of how observers utilize the information, ensuring flexibility.
- Scalability: The system can seamlessly integrate more observers—such as those for logging, alerts, and reports—without necessitating any alterations to the subject.
- Reusability: Both observers and subjects can be effectively reused in diverse contexts, enhancing overall efficiency.

Challenges and Solutions

Distinguishing Entry from Exit Events

The system confidently identifies whether a ParkingEvent represents a vehicle entering or exiting the lot, recognizing the significant differences in the responses for each scenario (e.g., checking in a permit versus finalizing a transaction). This necessity is effectively managed by incorporating a boolean field named isEntry into the ParkingEvent class. Upon receiving an event, the ParkingObserver decisively uses this flag to cache the timestamp for entries or to compute the parking duration and notify the TransactionManager for exits. An internal Map<Permit, LocalDateTime> efficiently manages the pairing of entries and exits.

Preventing Invalid or Duplicate Transactions

To ensure the integrity of our transaction logic, it is imperative to address situations where a vehicle re-enters before completing its exit or attempts to exit without a prior entry. The ParkingObserver rigorously validates each event through well-defined checks: if an entry is already recorded for a permit, the new entry is either ignored or flagged. Versus, the event is logged if an exit occurs without an associated entry, ensuring no transaction is generated. I had to find implemented debugging strategies, such as system.println(), to see where my code printed duplicate transactions. This systematic approach strengthens the system's robustness and aligns seamlessly with real-world parking management expectations.

Ensuring Functional Observer Notification Logic

As the observer logic operates in the background, verifying the accuracy of notifications presented a challenge during initial testing phases. I proactively addressed this by implementing JUnit tests to confirm several critical aspects: the creation of a ParkingTransaction following an entry-exit sequence, the simulation of multiple observers to guarantee full notification, and the validation of exception handling for edge cases (such as unregistered lots and invalid timestamps). This comprehensive testing framework not only enhances the system's reliability but also enables the swift identification of potential bugs during development.

Conclusion

Implementing the Observer pattern in the University Parking System has significantly enhanced its modularity, testability, and scalability. By effectively decoupling parking events from their corresponding actions, the system can evolve—such as by incorporating analytics or alert systems—without altering the core business logic. With accurate design and consideration of edge cases, the final solution aligns well with the principles of the Observer pattern and successfully emulates a real-world parking system.

References

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.

GeeksForGeeks. 2016a. “Observer Pattern | Set 1 (Introduction).” GeeksforGeeks. April 4, 2016. https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/.

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