Portfolio Assignment:

Implement Dependency Injection in the Parking System Application for

Master of Science

Information Technology

Bria Wright

University of Denver University College

June 8, 2025

Faculty: Nathan Braun MBA, BBA

Dean: Michael J. McGuire, MLS

Table of Contents

Introduction

The Parking System Application developed throughout the ICT-4315 course is a modular, real-time Java-based solution that simulates parking management. Key features include client-server communication via JSON, dynamic parking charge calculation, concurrent client handling, and data serialization for persistent storage. The application is designed to support the complete lifecycle of customer and car registration, permit issuance, parking events, and transaction logging.

A cornerstone of the project's architecture is its use of Dependency Injection (DI), a software design pattern that facilitates decoupling dependencies between components. DI enables greater flexibility, enhances testability, and supports code reuse (Martin 2024). In this project, Google Guice was selected as the DI framework due to its lightweight configuration, compile-time validation, and ease of integration in a Java ecosystem (Baeldung et al. 2024). Guice allowed the injection of appropriate ParkingChargeStrategy or ParkingChargeCalculator implementations, depending on the context, without altering core logic.

System Design: Pattern Analysis

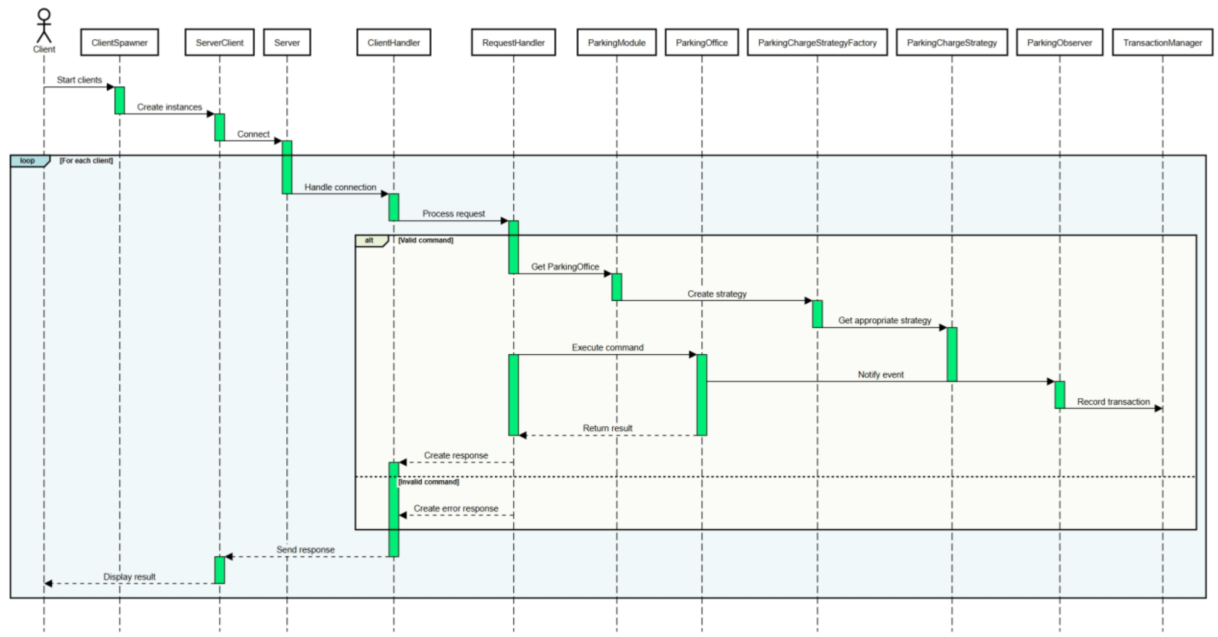

Figure 1: UML Sequence Diagram Updated

In Figure 1, the upated sequence diagram, shows how a parking management system handles client requests, calculates fees with flexible strategies, records transactions, and provides results in a multi-client setup. This sequence diagram shows how a parking management system works between clients and the server. It explains how clients connect, how requests are processed, how parking fees are calculated, and how notifications are sent (Gamma et al. 1994). Here's a simple breakdown:

- Client Startup
    - The client launches several instances using the ClientSpawner.
    - ServerClient instances connect to the Server.
- Server Handling
    - The Server assigns each client connection to a ClientHandler.
    - The ClientHandler sends the request to a RequestHandler.

- Processing Requests
    - Valid Commands:
        - The RequestHandler connects to the ParkingModule to get the ParkingOffice.
        - The ParkingModule uses the ParkingChargeStrategyFactory to choose a parking fee strategy.
        - The RequestHandler runs the command in the ParkingOffice.
        - The ParkingOffice informs the ParkingObserver about parking events (entry/exit).
        - The ParkingObserver records the transaction with the TransactionManager.
        - The result goes back to the client through the ClientHandler and ServerClient.
    - Invalid Commands:
    - The RequestHandler sends an error response to the client.
- Client Response
    - The ServerClient shares the result (success or error) with the Client.
- Key Interactions
    - Multi-Client Support: The ClientSpawner manages multiple ServerClient instances.
    - Dynamic Strategy Selection: The ParkingChargeStrategyFactory picks the right pricing strategy.

    - Event-Driven System: The ParkingObserver responds to parking events and logs transactions.

- Core Features
    - Scalability: The system supports many clients at the same time.
    - Flexible Pricing: Pricing strategies (like hourly or flat rate) can change as needed.
    - Audit Trail: All parking events are recorded by the TransactionManager.

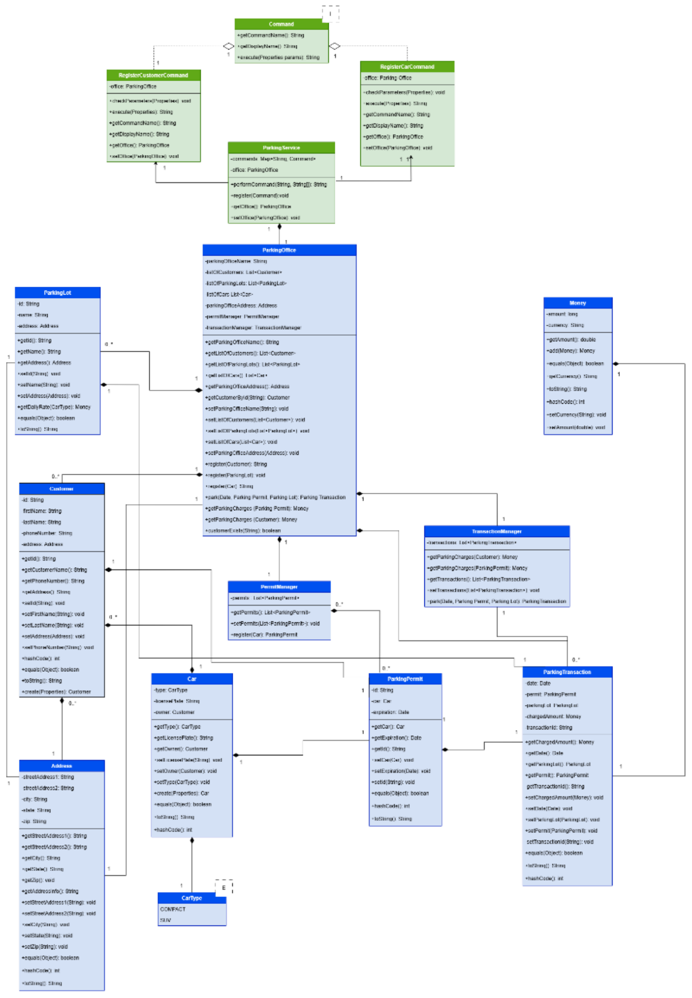
Figure 2: UML Class Diagram – ict4315.assigment_1 package

This structure emphasizes clean separation of responsibilities and strong type safety, aligned with principles from Clean Code (Martin 2008). The command pattern implementation (e.g., RegisterCarCommand) aligns with the behavioral design patterns advocated in Head First Design Patterns (Freeman 2020):

- *Customer*: Registered user with unique ID, manages multiple ParkingPermits
- *ParkingPermit*: Authorization to park specific Car in designated lots (1:1 with Car, 1:many with Customer)
- *Car*: Vehicle data (make/model/plate) linked to permits (1:many with ParkingPermit)

Parking Management

- *ParkingLot*: Contains pricing rules and processes transactions (1:many with ParkingTransaction)
- *ParkingOffice*: Main coordinator for registrations, permits, and transactions
- *PermitManager*: Issues/validates permits between Cars and ParkingLots

Financial System

- *Money*: Immutable currency handler with safe arithmetic operations
- *ParkingTransaction*: Records parking events with timestamps/fees
- *TransactionManager*: Maintains transaction integrity and history

Command Pattern

- *Command* (Abstract): Interface for all operations (execute())
- *RegisterCustomerCommand*: Creates new Customer entries
- *RegisterCarCommand*: Handles car/permit registration with validation

Supporting Classes

- *Address*: Customer address data container
- *CarType* (Enum): Vehicle classification (COMPACT & SUV)

- *ParkingService*: Command router and system entry point

Key Relationships

- Customers own Permits which reference Cars
- ParkingLots process Transactions managed by TransactionManager
- Commands modify system state via ParkingOffice
- Money objects used for all financial calculations

This structure demonstrates:

1. Clear entity ownership (Customers→Permits→Cars)
2. Separation of financial operations (Money/Transactions)
3. Extensible command pattern for operations
4. Central coordination through ParkingOffice
5. Type safety via CarType enum

The design emphasizes loose coupling, with ParkingService acting as the primary facade for system operations.

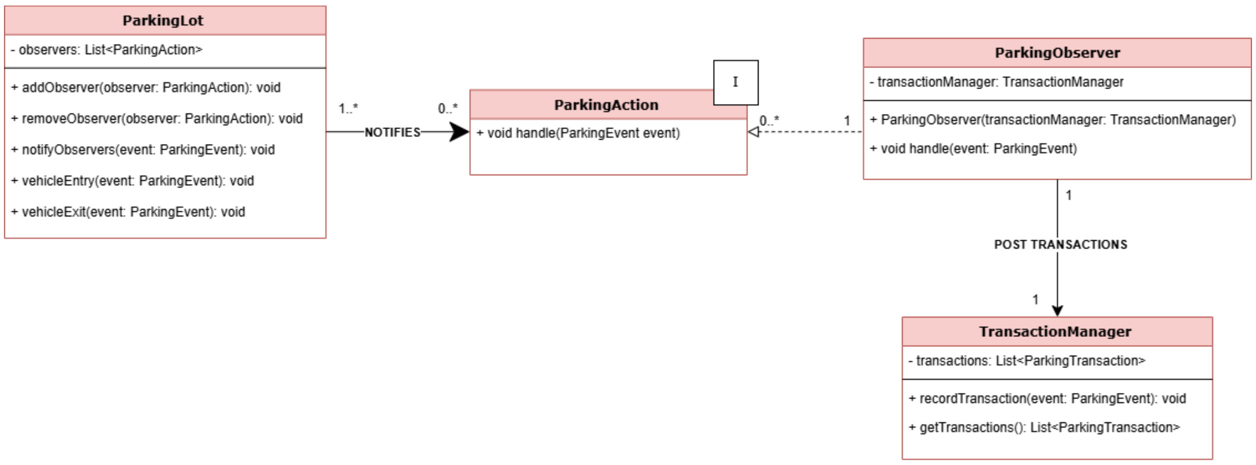
ParkingLot
- observers: List<ParkingAction>
+ addObserver(observer: ParkingAction): void
+ removeObserver(observer: ParkingAction): void
+ notifyObservers(event: ParkingEvent): void
+ vehicleEntry(event: ParkingEvent): void
+ vehicleExit(event: ParkingEvent): void
1..*
NOTIFIES
0..*
ParkingAction
I
+ 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 3: UML Class Diagram - Observers

I created a UML class diagram focusing on the new attributes and methods contributing to the ParkingObserver class. The observer implementation in ParkingObserver adheres closely to the classic Observer Pattern definition (Gamma et al. 1994), where ParkingLot acts as the subject, and ParkingObserver responds to parking events. This design ensures event tracking is decoupled from the event producer. 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.

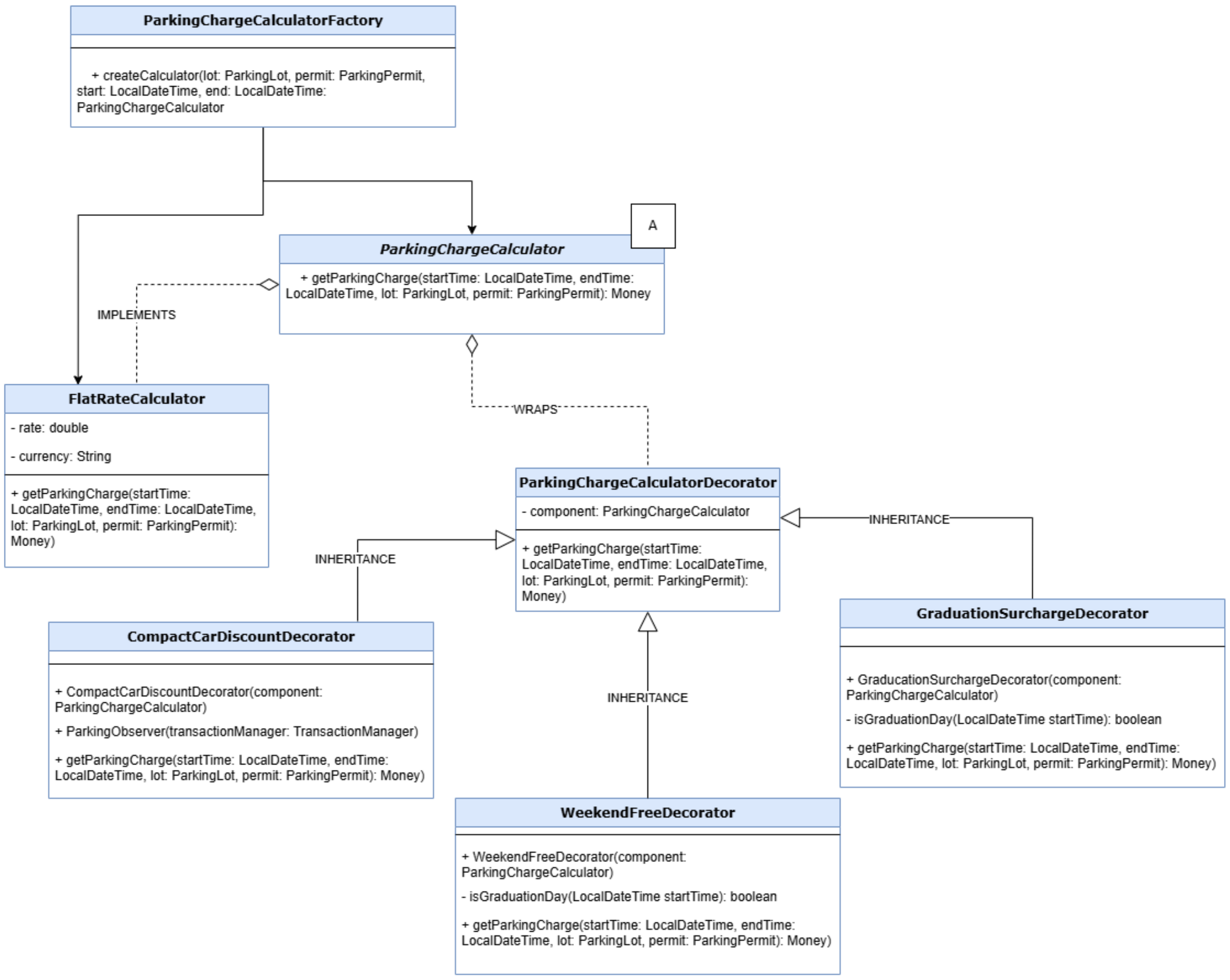
ParkingChargeCalculatorFactory
+ createCalculator(lot: ParkingLot, permit: ParkingPermit, start: LocalDateTime, end: LocalDateTime: ParkingChargeCalculator
A
ParkingChargeCalculator
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money
IMPLEMENTS
WRAPS
FlatRateCalculator
- rate: double
- currency: String
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money)
ParkingChargeCalculatorDecorator
- component: ParkingChargeCalculator
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money)
INHERITANCE
INHERITANCE
INHERITANCE
CompactCarDiscountDecorator
+ CompactCarDiscountDecorator(component: ParkingChargeCalculator)
+ ParkingObserver(transactionManager: TransactionManager)
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money)
GraduationSurchargeDecorator
+ GraducationSurchargeDecorator(component: ParkingChargeCalculator)
- isGraduationDay(LocalDateTime startTime): boolean
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money)
WeekendFreeDecorator
+ WeekendFreeDecorator(component: ParkingChargeCalculator)
- isGraduationDay(LocalDateTime startTime): boolean
+ getParkingCharge(startTime: LocalDateTime, endTime: LocalDateTime, lot: ParkingLot, permit: ParkingPermit): Money)

Figure 4: UML Class Diagram – Decorators

Decorators such as CompactCarDiscountDecorator dynamically wrap a base calculator and modify behavior—an implementation directly inspired by the Decorator Pattern (Gamma et al. 1994). This supports modular, runtime composition of behaviors without changing existing code, a principle emphasized in Clean Code (Martin 2008).The class diagram, Figure 4, (see attached) visually represents how decorators 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. Wright 6 • Decorators such as CompactCarDiscountDecorator extend ParkingChargeCalculator and hold a reference to another calculator, enabling recursive delegation of charge calculations.
- ParkingChargeCalculatorFactory constructs and returns a correctly ordered chain of decorators based on the parking session's context (e.g., permit type, vehicle size, date, and time), encapsulating the composition logic and promoting consistency.

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).

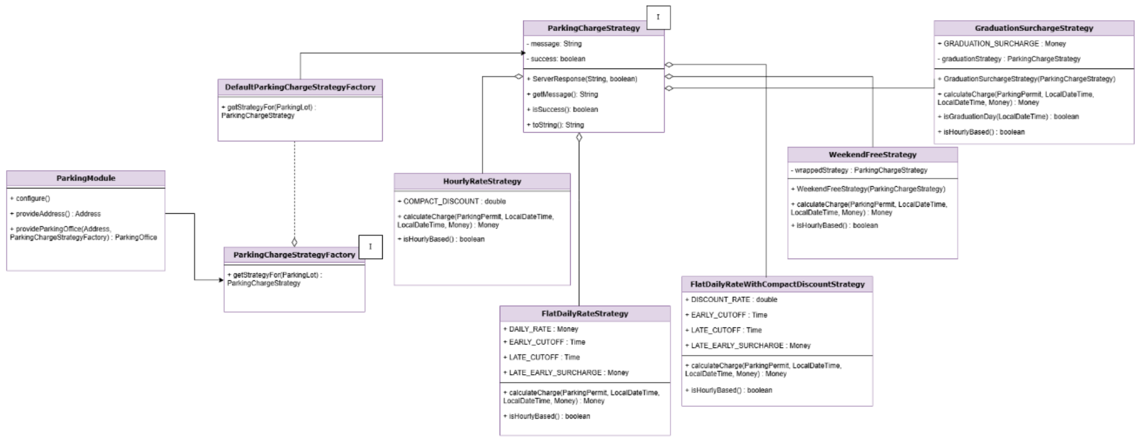
ParkingChargeStrategy
I
- message: String
- success: boolean
+ ServerResponse(String, boolean)
+ getMessage(): String
+ isSuccess(): boolean
+ toString(): String
GraduationSurchargeStrategy
+ GRADUATION_SURCHARGE : Money
- graduationStrategy : ParkingChargeStrategy
+ GraduationSurchargeStrategy(ParkingChargeStrategy)
+ calculateCharge(ParkingPermit, LocalDateTime, LocalDateTime, Money) : Money
+ isGraduationDay(LocalDateTime) : boolean
+ isHourlyBased() : boolean
DefaultParkingChargeStrategyFactory
+ getStrategyFor(ParkingLot) : ParkingChargeStrategy
WeekendFreeStrategy
- wrappedStrategy : ParkingChargeStrategy
+ WeekendFreeStrategy(ParkingChargeStrategy)
+ calculateCharge(ParkingPermit, LocalDateTime, LocalDateTime, Money) : Money
+ isHourlyBased() : boolean
ParkingModule
+ configure()
+ provideAddress() : Address
+ provideParkingOffice(Address, ParkingChargeStrategyFactory) : ParkingOffice
HourlyRateStrategy
+ COMPACT_DISCOUNT : double
+ calculateCharge(ParkingPermit, LocalDateTime, LocalDateTime, Money) : Money
+ isHourlyBased() : boolean
ParkingChargeStrategyFactory
I
+ getStrategyFor(ParkingLot) : ParkingChargeStrategy
FlatDailyRateWithCompactDiscountStrategy
+ DISCOUNT_RATE : double
+ EARLY_CUTOFF : Time
+ LATE_CUTOFF : Time
+ LATE_EARLY_SURCHARGE : Money
+ calculateCharge(ParkingPermit, LocalDateTime, LocalDateTime, Money) : Money
+ isHourlyBased() : boolean
FlatDailyRateStrategy
+ DAILY_RATE : Money
+ EARLY_CUTOFF : Time
+ LATE_CUTOFF : Time
+ LATE_EARLY_SURCHARGE : Money
+ calculateCharge(ParkingPermit, LocalDateTime, LocalDateTime, Money) : Money
+ isHourlyBased() : boolean

Figure 5: UML Class Diagram – Strategies

This UML diagram in Figure 6, represents a parking charge system using dependency injection (Guice) and the strategy pattern for calculating parking fees. This matches the canonical strategy design recommended by Gamma et al. (1994), further supported by dependency injection best practices discussed by Baeldung et al. (2017). Here's a breakdown:

- Guice Module (ParkingModule)
  - Configures dependency bindings.

  - Provides instances of Address and ParkingOffice.
  - Depends on ParkingChargeStrategyFactory.
- Parking Charge Strategy Factory (ParkingChargeStrategyFactory)
  - Interface: Defines a method (getStrategyFor) to retrieve a parking charge strategy based on the ParkingLot.
  - Default Implementation (DefaultParkingChargeStrategyFactory): Supplies the appropriate strategy.
- Parking Charge Strategies (ParkingChargeStrategy Implementations)
  - HourlyRateStrategy: Charges are based on hourly rates with a discount for compact cars.
  - FlatDailyRateStrategy: Applies a fixed daily rate with surcharges for early/late parking.
  - FlatDailyRateWithCompactDiscountStrategy: Similar to Flat Daily, but includes a discount for compact cars.
  - GraduationSurchargeStrategy: Adds a surcharge on graduation days, wrapping another strategy.
  - WeekendFreeStrategy: Makes parking free on weekends, delegating to another strategy otherwise.
- Key Relationships
  - Strategies implement the ParkingChargeStrategy interface.
  - GraduationSurchargeStrategy and WeekendFreeStrategy decorate other strategies (composition).

- The factory (DefaultParkingChargeStrategyFactory) selects and provides the appropriate strategy.

Core Concepts:

- Strategy Pattern: Different pricing algorithms are encapsulated as interchangeable strategies, allowing for easy modification and replacement.
- Dependency Injection (Guice): Manages object creation and dependencies.
- Decorator Pattern: Some strategies enhance others (e.g., adding weekend free logic or graduation surcharges).

This design enables flexible and maintainable parking fee calculations with varying business rules.

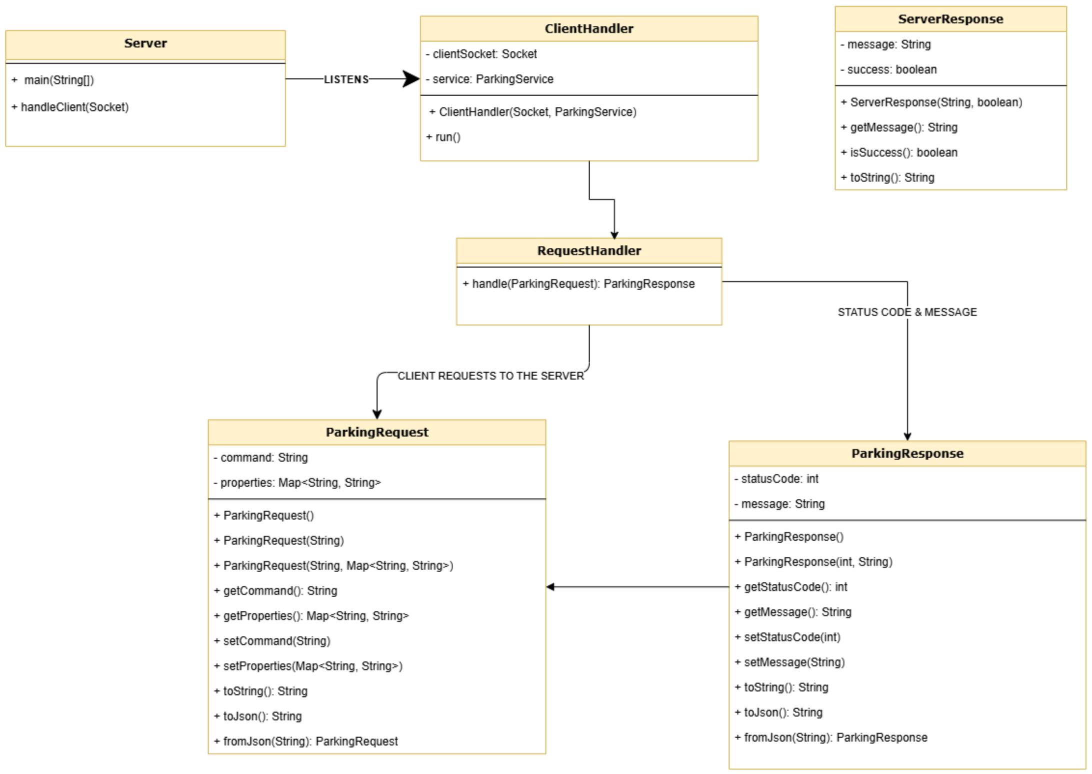

Figure 6: UML Class Diagram – Server/Requests

In this Figure 6, system facilitates scalable, JSON-based communication between parking clients and a multi-threaded server, ensuring a well-defined separation of networking, protocol, and business logic. The following UML diagram illustrates a client-server parking system characterized by request-response communication, JSON serialization, and multi-threaded client handling. Here's a breakdown of its components:

- Parking Protocol (Request & Response)
    - ParkingRequest: Represents a client request sent to the server.
    - Contains:
        - A command (e.g., "register_car," "calculate_fee").
        - A properties map (key-value pairs for request data).
    - Supports JSON serialization (toJson(), fromJson()).
- ParkingResponse: Represents the server's reply.
    - Contains:
        - A status code (e.g., for success & for errors).
        - A message (e.g., "Successfully registered").
    - It also supports JSON serialization.

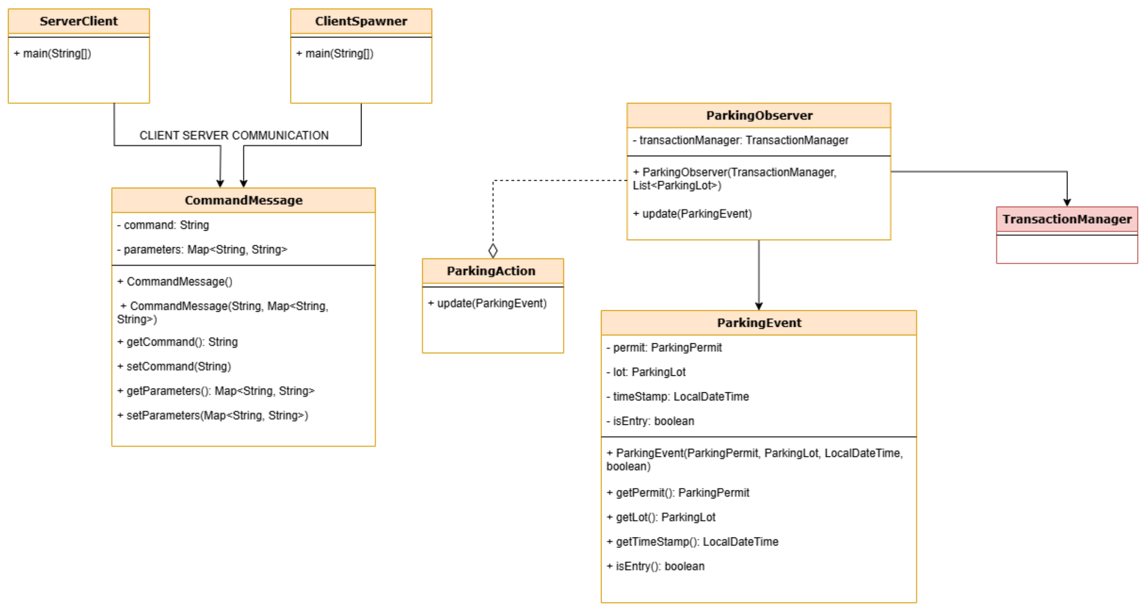
ServerClient
+ main(String[])
ClientSpawner
+ main(String[])
CLIENT SERVER COMMUNICATION
CommandMessage
- command: String
- parameters: Map<String, String>
+ CommandMessage()
+ CommandMessage(String, Map<String, String>)
+ getCommand(): String
+ setCommand(String)
+ getParameters(): Map<String, String>
+ setParameters(Map<String, String>)
ParkingObserver
- transactionManager: TransactionManager
+ ParkingObserver(TransactionManager, List<ParkingLot>)
+ update(ParkingEvent)
TransactionManager
ParkingAction
+ update(ParkingEvent)
ParkingEvent
- permit: ParkingPermit
- lot: ParkingLot
- timeStamp: LocalDateTime
- isEntry: boolean
+ ParkingEvent(ParkingPermit, ParkingLot, LocalDateTime, boolean)
+ getPermit(): ParkingPermit
+ getLot(): ParkingLot
+ getTimeStamp(): LocalDateTime
+ isEntry(): boolean

Figure 7: UML Class Diagram – Client Server Communication

This system helps manage parking using a client-server model with event-driven observation, in Figure 8. Clients send commands, and observers track parking actions in real time. The UML diagram below describes the parking system, focusing on communication, event observation, and parking action tracking (Freeman 2020).

- Client Components
    - ClientSpawner: Launches multiple clients for testing and sends commands to the server using CommandMessage.
- ServerClient: Represents a single client that interacts with the server.
    - Uses CommandMessage for communication.
- CommandMessage
    - A standard format for messages between clients and the server.
    - Includes:
        - A command (like "park" or "exit").

        - A parameters map with key-value information, such as permit ID or lot number.

- Observer Pattern
    - ParkingAction (Interface):
        - Defines an update(ParkingEvent) method for handling events.
    - ParkingObserver (Implementation):
        - Listens for ParkingEvent actions like entry and exit.
        - Uses TransactionManager to log or process events.
        - Tracks the status of the ParkingLot.
    - ParkingEvent:
        - Represents a parking entry or exit.
    - Includes:
        - ParkingPermit (the person who parked).
        - ParkingLo (the location of parking).
        - LocalDateTime (timestamp).
        - isEntry (true for entry, false for exit).
- Key Relationships
    - Clients → CommandMessage: Clients send requests using CommandMessage.
    - ParkingObserver → ParkingEvent: Observers respond to parking actions.
    - ParkingObserver → TransactionManager: Observers log or process events.
- Observer Pattern:
    - ParkingObserver implements the ParkingAction interface.

    - ParkingEvent triggers updates in observers.
- Core Features:
    - Client-Server Messaging: Uses CommandMessage for clear communication.
    - Event-Driven Parking Tracking: ParkingEvent records when cars enter or exit, while ParkingObserver monitors these events.
    - Extensibility: New observers can be added through the ParkingAction interface.

Lessons Learned

This project has significantly deepened my understanding of software architecture and clean code principles. I gained a solid appreciation for the SOLID principles (Martin 2008), particularly the concepts of Single Responsibility and Dependency Inversion, which guided separation between parking logic, transaction logging, and user interface commands. Here are some key areas where I've grown:

- Object-Oriented Principles: I gained a solid appreciation for the SOLID principles (Martin 2024), particularly the concepts of single responsibility and dependency inversion. These principles guided the clear separation of concerns among parking logic, transaction management, and user interface commands.
- Dependency Injection with Guice: Although initially unfamiliar, I learned how DI promotes decoupling and allows for flexible configurations. Guice's capacity to inject test doubles, such as mock ParkingChargeCalculatorFactory, made unit testing much more straightforward (Baeldung 2017).

- Unit Testing: I honed my skills in creating parameterized and mock-based tests. Testing with Guice allowed for clean setup and teardown processes, resulting in tests that were more deterministic and maintainable.
- Design Patterns: I implemented Strategy, Decorator, Command, and Observer patterns, each serving a specific role in architecture. Their strategic use helped reduce rigidity and enhance extensibility (Gamma et al. 1994).

Reflections on Improvements

If I had the chance to restart the project, I would take advantage of certain design practices from the very beginning:

- Early Use of DI: Guice was introduced mid-project, requiring some refactoring. Introducing DI from the start would have reduced coupling in the early stages and simplified testing.
- Error Handling: At first, exception handling was sparse. As the project progressed, I incorporated structured logging and clearer exception messages, which were invaluable during concurrency debugging. Adopting a more standardized logging framework, such as SLF4J, would enhance this aspect even further.
- Concurrency Edge Cases: While I implemented multithreading through per-client threads, I identified potential edge cases like simultaneous writes or race conditions in shared structures (e.g., activeEntries).

Additionally, despite the effectiveness of the Decorator and Strategy patterns individually, managing both simultaneously proved complex. A more integrated charge calculation pipeline could boost clarity.

Future Enhancements

With more time or for a production scenario, I would implement:

- Web UI or REST API Layer: Replacing the socket-based client with a REST API would allow easier integration with web/mobile apps. Frameworks like Spring Boot or Javalin could handle HTTP requests (Spring 2019) ("Documentation - Javalin - a Lightweight Java and Kotlin Web Framework. Create REST APIs in Java or Kotlin Easily.," n.d.).
- Role-Based Permits: Introducing staff/student/public permits with time restrictions or zone-specific pricing would reflect real-world parking rules (Al-Turjman and Malekloo 2019).
- Advanced Scheduling: Implementing peak/off-peak pricing, holiday exemptions, or loyalty programs would make the system more robust and attractive to businesses (Ai, Simchi-Levi, and Zhu 2025) (Dorotic, Bijmolt, and Verhoef 2011).
- Rate Configuration UI: An admin dashboard for adjusting lot base rates, graduation dates, and discount policies would enable greater flexibility without necessitating code changes (Fahim, Hasan, and Chowdhury 2021) ("Creating UI | Vaadin Docs" 2022).

Conclusion

The Parking System Application has been transformed into a modular, maintainable, and testable solution, capable of managing dynamic parking needs in a multi-client setting. By integrating Guice for Dependency Injection, applying various object-oriented design patterns, and constructing a multithreaded server-client architecture, I was able to develop software that adheres to industry best practices.

I now feel more confident in designing systems that are both extensible and easy to test. This project taught me how well-structured architecture, effective design patterns, and proper dependency management can elevate messy and fragile code into something organized and robust. Moving forward, I will apply these lessons to any software I develop, particularly the significance of clean architecture and modularity.

References

Ai, Rui, David Simchi-Levi, and Feng Zhu. 2025. *Dynamic Service Fee Pricing under Strategic Behavior: Actions as Instruments and Phase Transition*. Curran Associates Inc. https://dl.acm.org/doi/10.5555/3737916.3739611.

Al-Turjman, Fadi, and Arman Malekloo. 2019. “Smart Parking in IoT-Enabled Cities: A Survey.” *Sustainable Cities and Society* 49, no. August (August): 101608. https://doi.org/10.1016/j.scs.2019.101608.

Baeldung. 2017. “Baeldung.” Baeldung. March 14, 2017. https://www.baeldung.com/guice.

“Creating UI | Vaadin Docs.” 2022. Vaadin.com. 2022. https://vaadin.com/docs/v23/create-ui.

“Documentation - Javalin - a Lightweight Java and Kotlin Web Framework. Create REST APIs in Java or Kotlin Easily.” n.d. Javalin. https://javalin.io/documentation.

Dorotic, Matilda, Tammo H.A. Bijmolt, and Peter C. Verhoef. 2011. “Loyalty Programmes: Current Knowledge and Research Directions*.” *International Journal of Management Reviews* 14, no. 3 (June): 217–37.

Fahim, Abrar, Mehedi Hasan, and Muhtasim Alam Chowdhury. 2021. “Smart Parking Systems: Comprehensive Review Based on Various Aspects.” *Heliyon* 7, no. 5 (May): e07050. https://doi.org/10.1016/j.heliyon.2021.e07050.

Freeman, Eric . 2020. “Head First Design Patterns, 2nd Edition.” Edited by Elisabeth Robson. O’Reilly Online Learning. 2020. https://learning.oreilly.com/library/view/head-first-design/9781492077992/.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. “Design Patterns: Elements of Reusable Object Oriented Software.” https://www.grch.com.ar/docs/unlu.poo/Gamma-DesignPatternsIntro.pdf.

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/.

Spring. 2019. “Spring Projects.” Spring.io. 2019. https://spring.io/projects/spring-boot.