Week 10 Assignment: Implement Dependency Injection in the Parking System for

ICT 4315-1 Object-Oriented Methods & Paradigms II

Jack Hermanson

University of Denver University College

June 8, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

Table of Contents

`

Introduction

This paper discusses the parking system overall, as well as the basic implementation of dependency injection using Guice in the parking system application.

Parking System

The parking system is made up of parking lots, which can be “scan on entry” and “scan on entry and exit” types. “Scan on entry” lots are akin to daily lots, where customers are charged for the full day upon entering and then charged again every morning that they remain in the lot. “Scan on exit” lots are akin to hourly lots, where customers are charged for the duration of time they spent in the lot. Customers use permits to track which cars have entered lots and generate transactions accordingly. The parking system makes a concurrent HTTP server available that accepts commands, such as “enter”, which modify the state of the application. Additionally, there are useful classes like Money and Address to help store data in an easily testable manner.

Dependency Injection

Because the application was built in a way that repeatedly modified the code to demonstrate different concepts (e.g., strategy pattern, factory pattern, decorator pattern, observer pattern, etc.), the code is not very tidy or consistent. To work around this without spending weeks refactoring, I decided to use dependency injection for three core pieces of functionality: registering a car, entering a lot, and exiting a lot. Dependency injection helped enable the application to have shared instances of objects across different HTTP requests. For example, if one request comes in that has a car enter a lot, the next request will be working

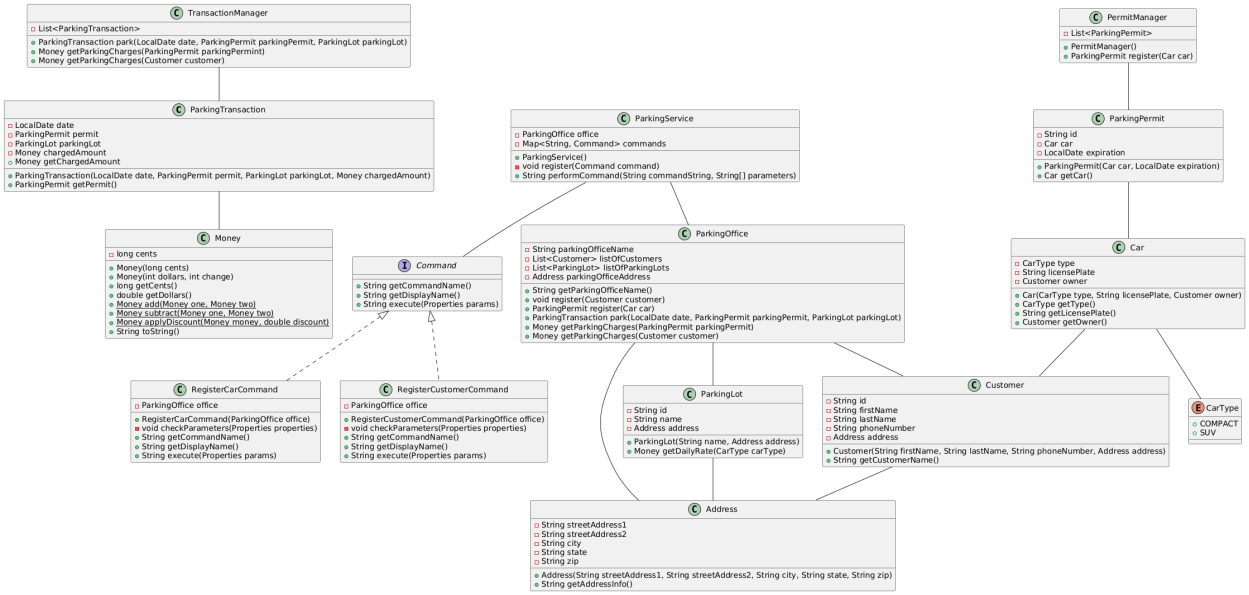
with the same exact instance of that lot, so the two requests will modify the same data. I will go into more details about this further in the paper.

System Design

UML Diagrams

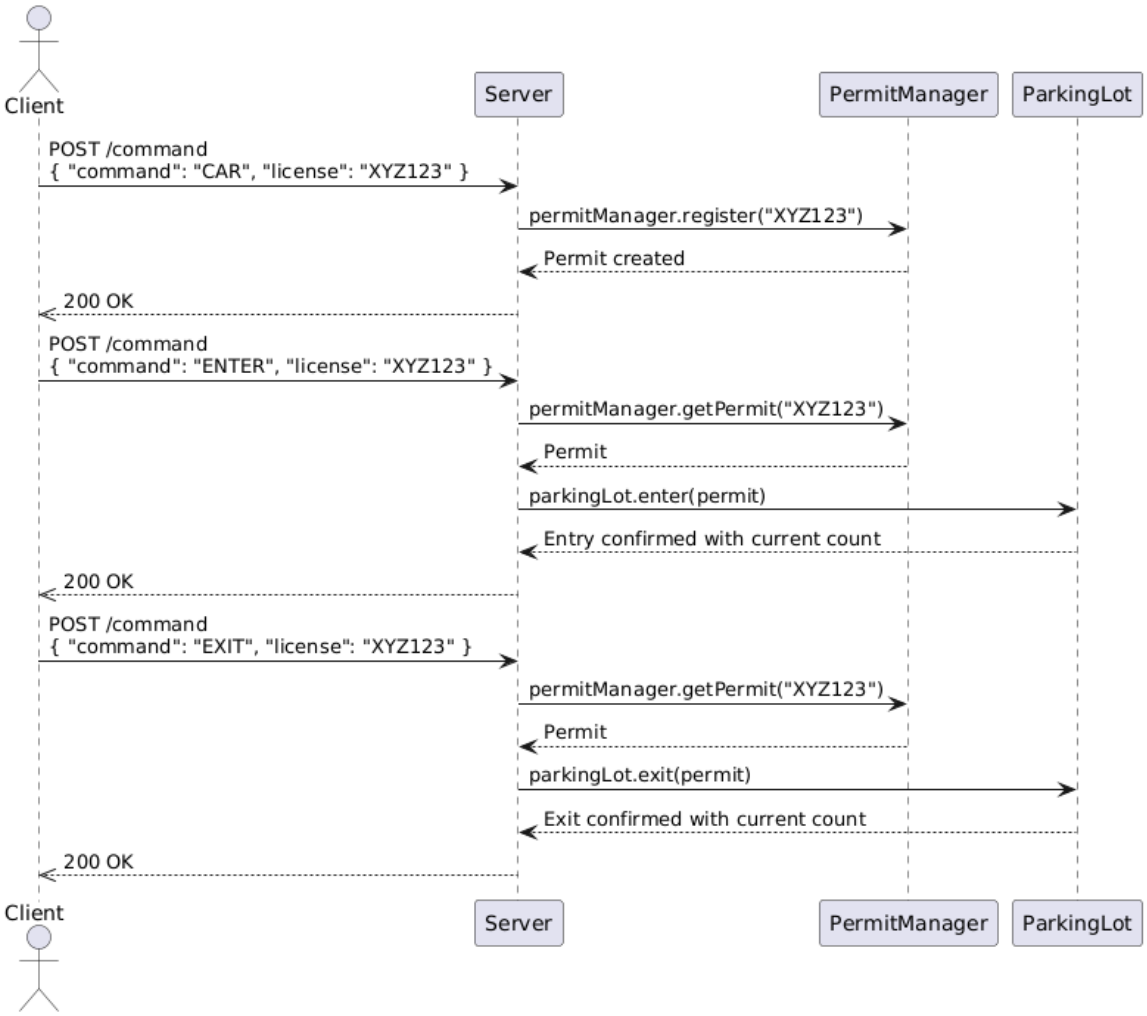
The following class diagram shows an overview of the system with some details omitted for brevity. You can see how different classes relate to one another.

Figure 1: High-Level Class Diagram



The following sequence diagram shows how the server modifies the state of the parking system based on requests it receives with different commands. We will see more details about this process when looking at dependency injection.

Figure 2: Sequence Diagram



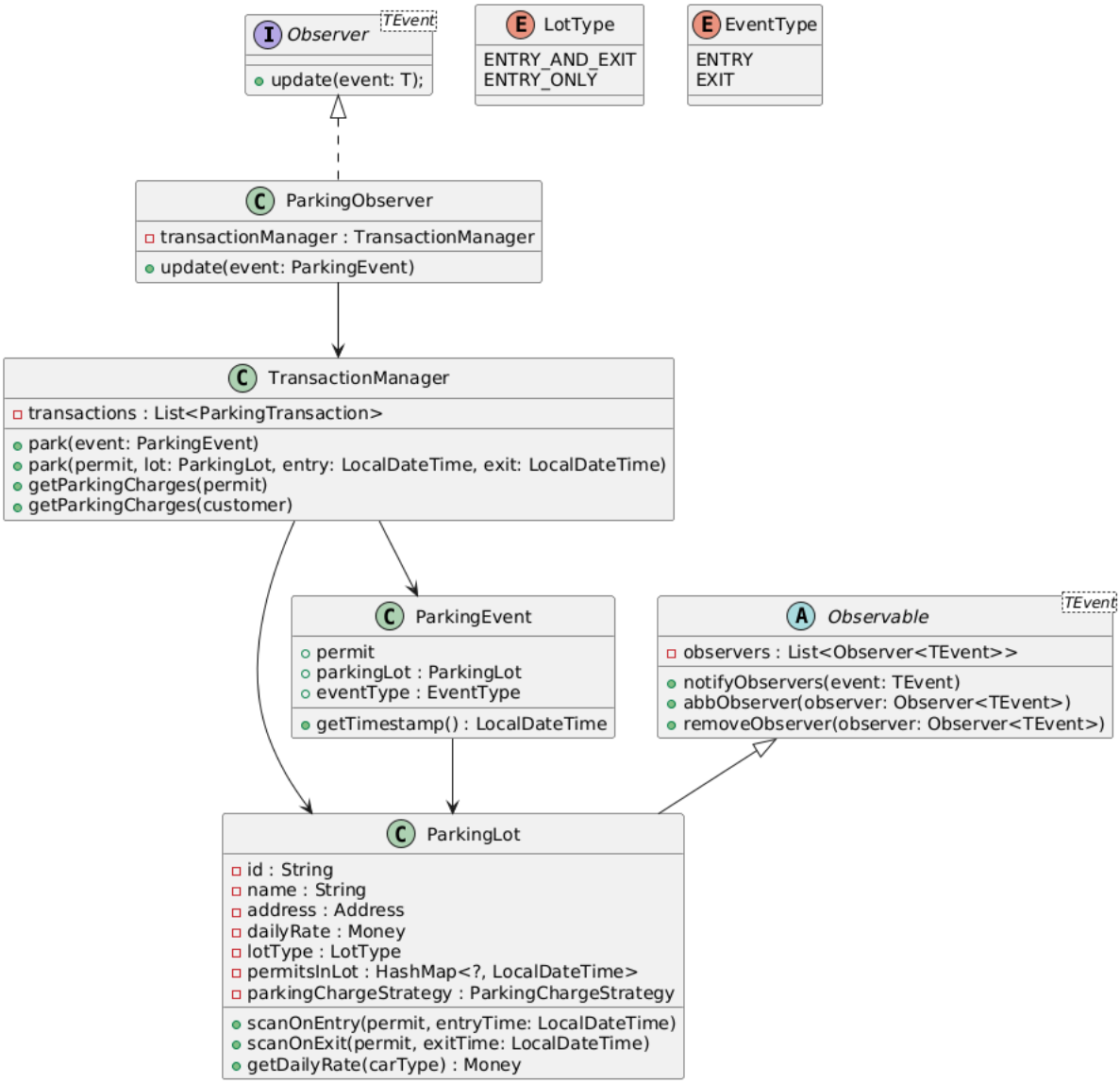
Object-Oriented Design

Various components of object-oriented design were used in this project. One such component is the Strategy Pattern complemented by the Factory Pattern. The Strategy Pattern simply defined an interface for generating transactions, given some set of parameters. Different scenarios require different strategies (with different internal logic) to generate a proper transaction. For example, “scan on entry” lots use different logic than “scan on entry and exit” lots, and special days like graduation have entirely different logic. The factory pattern allowed

me to create a “virtual constructor” for the different parking charge strategies that vary based on the lot type (Refactoring Guru 2025). The benefit of this is that, when it’s time to create a parking transaction, the transaction manager will use a factory to get the right type of strategy to produce a transaction with the correct amount charged. The transaction manager itself does not need to worry about the logic behind determining which strategy to use; the factory does this. The logic is abstracted away with any complexity.

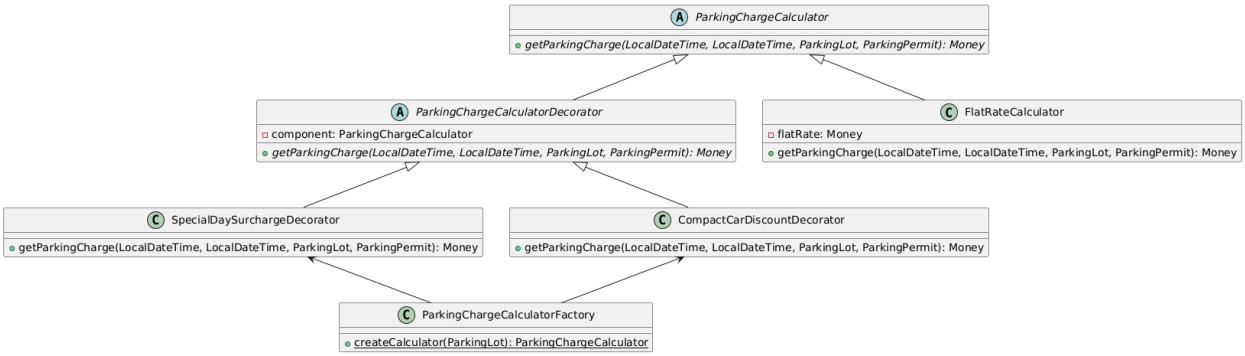
Another object-oriented design pattern is the Observer Pattern. The Observer Pattern, at a high level, involves two roles: publishers and subscribers (Refactoring Guru 2025). In the parking system, ParkingLot extends the Observable class, which means other classes can “observe” the parking lot for events that happen, such as vehicles entering and exiting. Observers must implement the Observer interface, which provides an update method so that observables can communicate with observers about things that happened. In the parking system, the transaction manager observes the parking lots for entry and exit so that it can generate transactions accordingly.

Figure 3: Observer Pattern Class Diagram



The third pattern used in the system is the decorator pattern, which adds functionality to objects by placing them inside special “wrapper” classes that contain these additional behaviors (Refactoring Guru, 2025).

Figure 4: Decorator Pattern



The application also uses the object-oriented design pattern of “dependency injection”, which I will discuss in the next section.

Lessons Learned

Several lessons I learned were already covered in the previous section that outlined different patterns and object-oriented principles. However, I learned more substantial lessons regarding concurrency and dependency injection.

Concurrency

With the introduction of the HTTP server to accept commands that modify the state of the application, it became necessary to handle multiple requests at once to avoid unnecessary slowness. For example, if two permits are being created, this doesn’t need to be done in sequence. Further, this process doesn’t need to block cars from entering and exiting lots. To solve this problem, I had the HTTP server run on 4 threads simultaneously, each able to handle a request independently of the other. However, multiple requests may come in that modify some data. To ensure that the data is accurate and stable, I used the ConcurrentHashMap class,

which allows “multiple threads to read and write data simultaneously, without the need for locking the entire map” (GeeksForGeeks 2025). This allows each parking lot to maintain a list of parking permits that are currently in the lot (mapped to their entry time) while also allowing multiple entrances so let cars in at the same time.

Dependency Injection

The introduction of the HTTP server also created an opportunity to use dependency injection. Guice allowed me to create instances of a class that persist across the entire runtime of the server without needing to use the “new” operator, because the instances are “injected” where they are needed (Googler 2021). When the server starts up, it configures three classes for dependency injection: ParkingChargeStrategy (using the EntryAndExitChargeStrategy implementation), PermitManager, and ParkingLot. This was useful because I did not need to manually pass around the chosen parking charge strategy, list of all known permits, or reference to a parking lot with a list of permits currently in it. Instead, these instances are created up front, and different classes can use them as needed through dependency injection.

This complemented the concurrency improvements because now all threads can request an instance of the parking lot, for example, in order to have a car enter or exit. One thread can register a car, the next can have it enter a lot, and the next can have it exit. Here’s what that looks like in action with some manual delays inserted:

Figure 5: Dependency Injection and Concurrency Output

```
Server is running on port 8000
[23:34:55] Handling request on thread: pool-1-thread-1
Permit created successfully for ROB4CO; permits count: 1
[23:34:55] Request has been completed on thread pool-1-thread-1
Took 0 seconds
[23:34:55] Handling request on thread: pool-1-thread-3
Delaying 5000ms
[23:34:56] Handling request on thread: pool-1-thread-4
Delaying 10000ms
ROB4CO entered the lot; number of cars in lot: 1
[23:35:00] Request has been completed on thread pool-1-thread-3
Took 5 seconds
ROB4CO exited the lot; number of cars in lot: 0
[23:35:06] Request has been completed on thread pool-1-thread-4
Took 10 seconds
```

```
{
  "statusCode": 200,
  "message": "Successfully registered car with license plate: ROB4CO"
}
```

```
{
  "statusCode": 200,
  "message": "Car with license plate ROB4CO successfully entered lot"
}
```

```
{
  "statusCode": 200,
  "message": "Car with license plate ROB4CO successfully exited lot"
}
```

As you can see in the output, three separate requests were made, each was handled on its own thread, and each operation was seamless. Without dependency injection, it would be complicated to try to make sure all parts of the app are working with the same instance of the

ParkingLot class, for example, creating the potential for a request to “exit” the lot to fail because it was not aware that the car had ever entered.

Reflections on Improvements

I think this application was sued to try to demonstrate too many different concepts, and it suffered as a result. It would have been better to have smaller applications to teach different ideas. There are too many different things going on in the application, making it difficult to understand and maintain. If I were to start over, I would use dependency injection from the beginning, with coordination at the top level, rather than the current tangled mess of classes and unused properties that were created to demonstrate a concept each week that was not necessarily compatible with the following week’s lesson.

Another challenge came with the strategy pattern. The different types of lots (“entry and exit” vs “entry only”) are so different that the strategy pattern did not make sense, and I would not use it. The reason it didn’t make sense is that “entry and exit” lots require the entry and exit times to generate a transaction, while the “entry only” lots, by definition, do not have access to the exit time. As a result, I had to pass “null” to the exit parameter for “entry only” lots. There might be ways to deal with this while still using the factory pattern, however.

Future Enhancements

As a future enhancement, I would add more advanced unit testing. While my current unit testing gets the job done, I think dependency injection with interfaces facilitates the use of “mocks” to mock auxiliary behavior so you can test the functionality of what a given unit test is actually trying to test. For example, rather than having to create a fully functioning permit

manager in order to test entry and exit from a lot, you could “mock” the functionality of the dependency-injected permit manager and focus instead on the behavior of the parking lot.

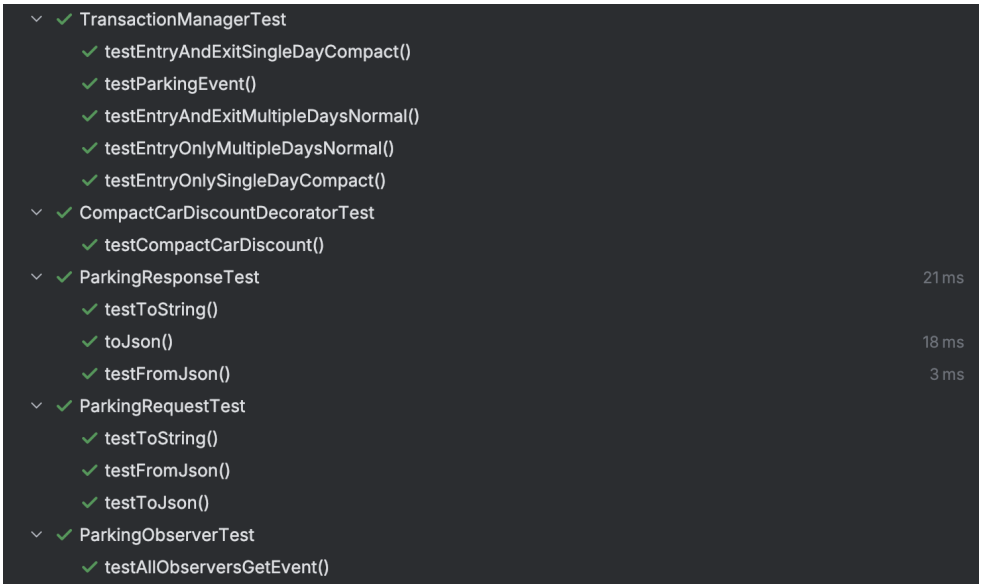
I might also add more threads if the server is capable of handling them so that the application can scale more easily.

Conclusion

In conclusion, the parking system application makes use of a wide range of design patterns, unit tests, concurrency, and other advanced object-oriented topics to achieve its goals. These concepts can be used together or separately depending on the situation. Dependency injection is a particularly useful tool for managing state across a large application. While the basic components work as intended (enough to demonstrate the key ideas without overcomplicating things), there is room for improvement in the testing and scalability of the application. Finally, with the larger picture now available at the end of the project, I might make different decisions about which patterns to use to accomplish important tasks.

Program Compiles and Tests Pass

The program builds successfully and all unit tests pass.



References

GeeksforGeeks. 2025. “ConcurrentHashMap in Java.” *GeeksforGeeks.* Last modified February 14, 2025. https://www.geeksforgeeks.org/concurrenthashmap-in-java/.

Googler. 2021. "Guice: Overview." GitHub. November 29, 2021. https://github.com/google/guice/wiki/.

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

Refactoring Guru. 2025. “Factory Method.” Accessed April 27, 2025. https://refactoring.guru/design-patterns/factory-method.

Refactoring Guru. 2025. “Observer.” Accessed April 30, 2025. https://refactoring.guru/design-patterns/observer.