Portfolio Assignment- Implement Dependency Injection in the Parking System Application

for

Master of Science

Information Technology

Peter Fedor

University of Denver College of Professional Studies

June 6th, 2025

Faculty: Nathan Braun, MS

Director: Cathie Wilson, MS

Dean: Michael J. McGuire, MLS

Introduction

Over the course of this quarter, we were tasked with creating a comprehensive parking lot management system for the university. The main purpose of the project was to apply object-oriented programming methods with Java programming language to create a robust and realistic system. As the weeks went by, additional requirements were added to the system. Some examples of added features include factory design patterns, observer design patterns, as well as dependency injections, which was this week's and the final addition to our system.

Design of the system

The system includes several main components, each responsible for a specific part of the parking application. Customer and Car classes represent the real-world entities being served by the parking office. ParkingPermit connects a car to a customer, and ParkingLot defines where vehicles can park. TransactionManager handles parking charges, while PermitManager is responsible for registering and tracking permits. One of the major shifts in the design was the use of the Strategy Pattern to allow different pricing models, such as daily, hourly, or event-based charges. This was a huge change as before, there was only a hard-coded flat rate for parking. This was implemented with strategy pattern originally. Later, we switched to the Decorator Pattern, which allowed dynamic combinations of pricing factors like compact car discounts or overnight fees. The system also uses the Observer Pattern, where the ParkingLot notifies observers when a car parks. I created an event object to encapsulate the details, which allowed observers—like the TransactionManager—to respond to parking events in real time. This was an important update to the system as well, as it adds separation between actions and logic within the system. It also allowed for actions such as logging and calculations to be made easier. In this past week, I added dependency injection through Google Guice. This required some structural changes, like defining interfaces for IPermitManager and

ITransactionManager, and creating a Guice module to wire the dependencies. The ParkingOffice class was then refactored to accept these interfaces via constructor injection, improving scalability, maintainability, and testability.

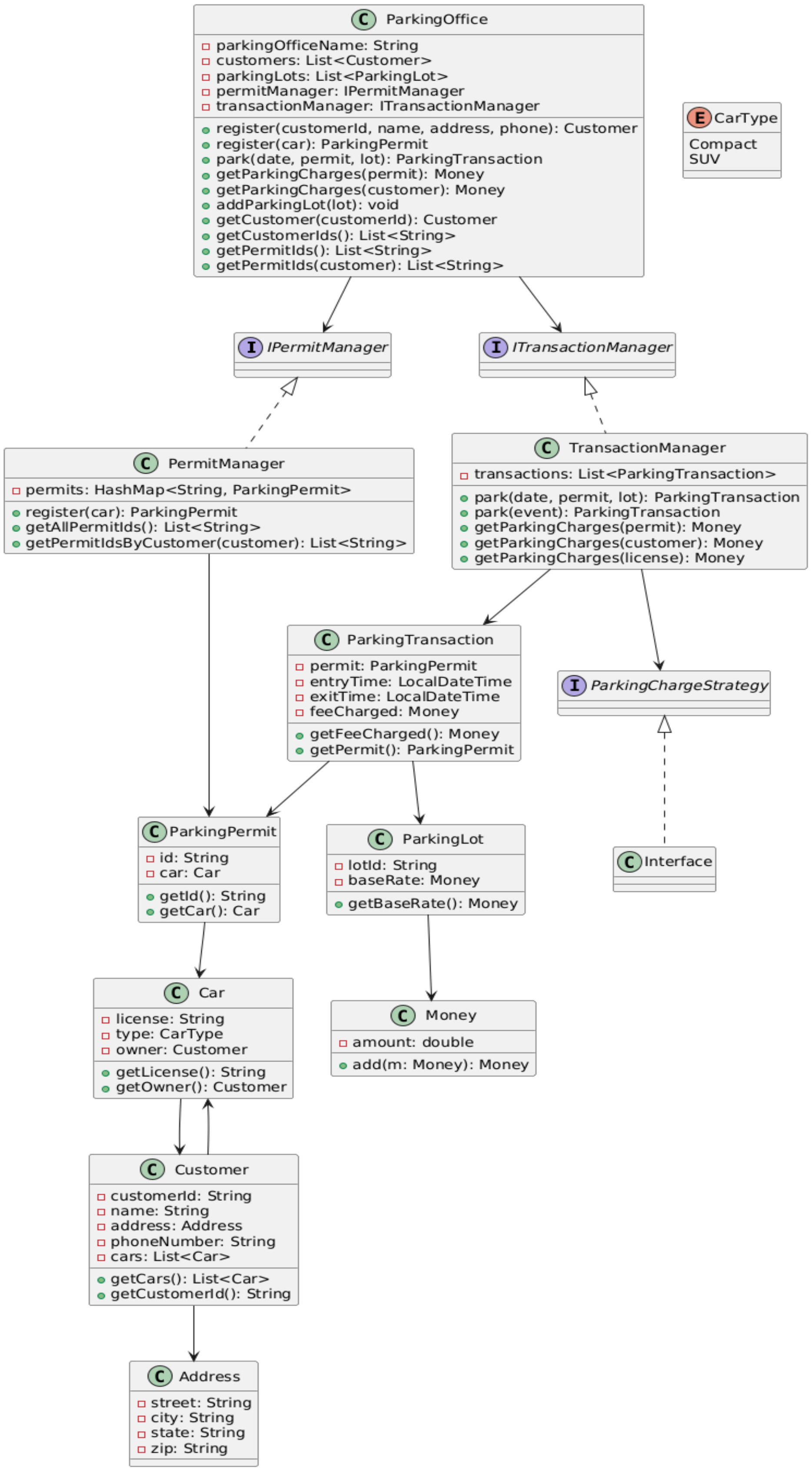
ParkingOffice
parkingOfficeName: String
customers: List<Customer>
parkingLots: List<ParkingLot>
permitManager: IPermitManager
transactionManager: ITransactionManager
register(customerId, name, address, phone): Customer
register(car): ParkingPermit
park(date, permit, lot): ParkingTransaction
getParkingCharges(permit): Money
getParkingCharges(customer): Money
addParkingLot(lot): void
getCustomer(customerId): Customer
getCustomerIds(): List<String>
getPermitIds(): List<String>
getPermitIds(customer): List<String>
CarType
Compact
SUV
IPermitManager
ITransactionManager
PermitManager
permits: HashMap<String, ParkingPermit>
register(car): ParkingPermit
getAllPermitIds(): List<String>
getPermitIdsByCustomer(customer): List<String>
TransactionManager
transactions: List<ParkingTransaction>
park(date, permit, lot): ParkingTransaction
park(event): ParkingTransaction
getParkingCharges(permit): Money
getParkingCharges(customer): Money
getParkingCharges(license): Money
ParkingTransaction
permit: ParkingPermit
entryTime: LocalDateTime
exitTime: LocalDateTime
feeCharged: Money
getFeeCharged(): Money
getPermit(): ParkingPermit
ParkingChargeStrategy
ParkingPermit
id: String
car: Car
getId(): String
getCar(): Car
ParkingLot
lotId: String
baseRate: Money
getBaseRate(): Money
Interface
Car
license: String
type: CarType
owner: Customer
getLicense(): String
getOwner(): Customer
Money
amount: double
add(m: Money): Money
Customer
customerId: String
name: String
address: Address
phoneNumber: String
cars: List<Car>
getCars(): List<Car>
getCustomerId(): String
Address
street: String
city: String
state: String
zip: String

Figure 1 UML Diagram

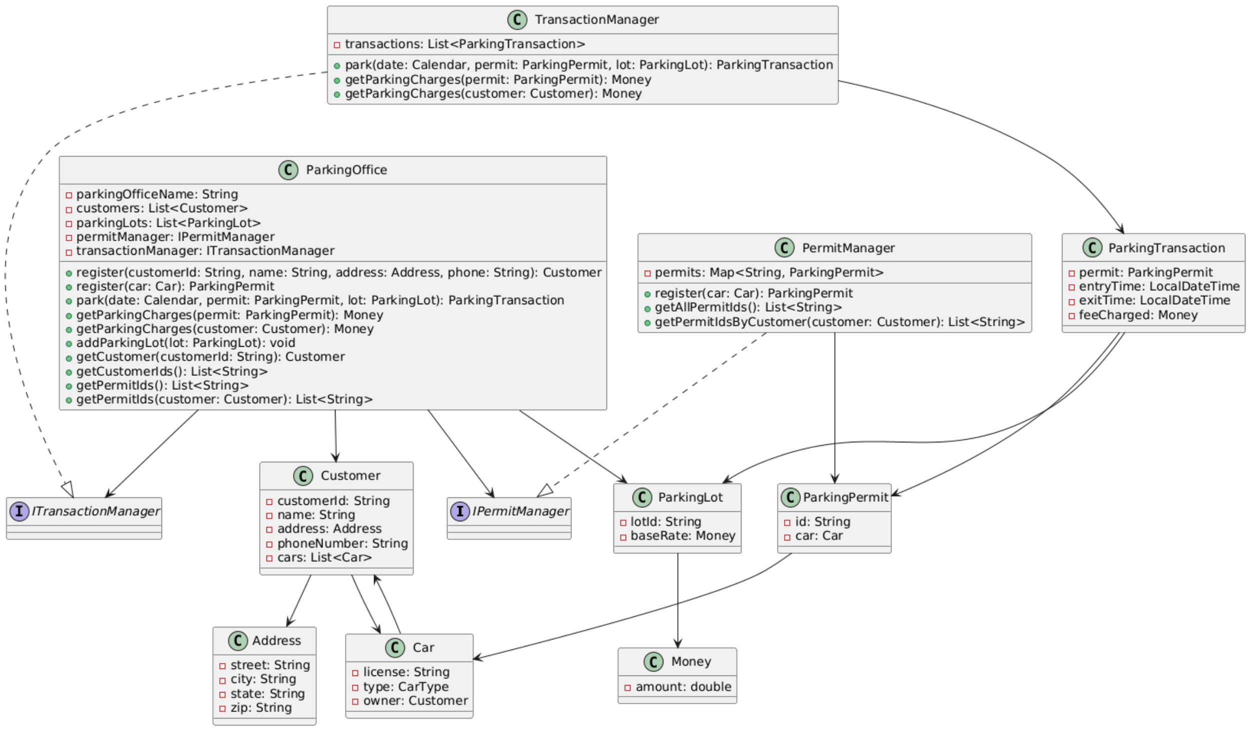
TransactionManager
- transactions: List<ParkingTransaction>
+ park(date: Calendar, permit: ParkingPermit, lot: ParkingLot): ParkingTransaction
+ getParkingCharges(permit: ParkingPermit): Money
+ getParkingCharges(customer: Customer): Money
ParkingOffice
- parkingOfficeName: String
- customers: List<Customer>
- parkingLots: List<ParkingLot>
- permitManager: IPermitManager
- transactionManager: ITransactionManager
+ register(customerId: String, name: String, address: Address, phone: String): Customer
+ register(car: Car): ParkingPermit
+ park(date: Calendar, permit: ParkingPermit, lot: ParkingLot): ParkingTransaction
+ getParkingCharges(permit: ParkingPermit): Money
+ getParkingCharges(customer: Customer): Money
+ addParkingLot(lot: ParkingLot): void
+ getCustomer(customerId: String): Customer
+ getCustomerIds(): List<String>
+ getPermitIds(): List<String>
+ getPermitIds(customer: Customer): List<String>
PermitManager
- permits: Map<String, ParkingPermit>
+ register(car: Car): ParkingPermit
+ getAllPermitIds(): List<String>
+ getPermitIdsByCustomer(customer: Customer): List<String>
ParkingTransaction
- permit: ParkingPermit
- entryTime: LocalDateTime
- exitTime: LocalDateTime
- feeCharged: Money
ITransactionManager
Customer
- customerId: String
- name: String
- address: Address
- phoneNumber: String
- cars: List<Car>
IPermitManager
ParkingLot
- lotId: String
- baseRate: Money
ParkingPermit
- id: String
- car: Car
Address
- street: String
- city: String
- state: String
- zip: String
Car
- license: String
- type: CarType
- owner: Customer
Money
- amount: double

Figure 2Class Diagram

Lessons Learned

One of the biggest lessons was understanding how design patterns can simplify system maintenance and flexibility. Initially, I tried hardcoding some logic, but as the system grew, I realized it became harder to maintain. Using patterns like Strategy and Decorator helped encapsulate behaviors and made it easier to swap in new logic without rewriting core code. Learning Guice was also a turning point. I had heard of dependency injection but hadn't implemented it in a real system. At first, it seemed like extra complexity, but after writing the Guice module and seeing how easily I could inject mock services during testing, it made sense. I also found that my unit tests became more focused and manageable. I became more confident with tools like JUnit for testing, PlantUML for diagrams, and even with troubleshooting run time errors in my IDE, Eclipse. I also learned about the importance of unit testing. Throughout the quarter, I often found myself thinking how useful it is to have Junit. Since most of my programming background is in Python, I did not know tools like unit tests existed.

As far as different tools and frameworks go, pretty much everything we had learned this quarter was new to me. One that I found particularly challenging to learn was Week 8, where we set up a server within our program, and used serialization to transmit information received. This specific assignment took me quite a bit of time to figure out. Specifically getting my server to run properly and return the response that the client was asking for. Another example is from this week's assignment implementing Guice into our system. I had to go back and figure out exactly what code I needed to change in order to get it to function properly. My approach to learning these examples, as well as the rest of the topics we covered, was to do research and review simple example codes given in websites like Baeldung and Geeks for Geeks. In the end, there were many challenging and new concepts I learned about this quarter. Coming into this class, I do wish that I was more confident with the Java language and object-oriented programming as a whole. At times I felt frustrated and lost when implementing new features to the parking lot system.

Reflections and Improvements

If I had to start over, I would have organized my code better. I had many classes that were not clearly named, making it hard to understand what each class was at a quick glance. I also would've structured my packages more cleanly from the start, dividing responsibilities more clearly between domain, service, and utility classes. Another improvement would be incorporating better exception handling. Right now, many methods return null or silently fail if inputs are incorrect. Adding better exception handling will help the system be more maintainable in the long run. I also wish I had started with the template code, instead of using what I had created in object oriented one. This is because my code was messy, and parts did not work. I think I would have had more time to write cleaner and better documented code if I wasn't scrambling to fix my mistakes from the prior quarter.

Throughout the course, I was faced with many new concepts that were really hard for me to understand. I found that watching YouTube videos, and using alternative resources helped me out quite a bit. One resource that stood out the most to me was Geeks for Geeks. The website had many valuable explanations of topics covered throughout the quarter. Another great resource was stack exchange. I specifically found troubleshooting code was made a lot easier with stack exchange since others have had similar issues with their own code before. It allowed me to understand why I was getting errors and allowed me to fix them myself.

Future Enhancements

With more time, I would have liked to add a user interface layer, possibly a simple GUI or a web front end. That would make the application more interactive and demonstrate a full parking lot system set up. This could be as simple as a portal for customers registering their vehicles or paying any charges they have incurred. Another enhancement would be to persist in customer, permit, and transaction data in a database. Right now, the system runs entirely in memory, so all data is lost between sessions. Integrating with a lightweight SQL database or even a flat file system would make the system more realistic. From a scalability perspective, I would also consider splitting parts of the system into separate services, maybe even a microservice for permit generation and another for charge processing. I still have a lot to learn about the capabilities of object oriented programming and Java, but these are areas that I feel can be improved upon in the application.

Conclusion

Overall, building the Parking System Application has been one of the most hands-on experiences I've had with object-oriented design. Through iterative development, I was able to take a basic design and evolve it into a modular, testable, and maintainable system. I implemented multiple design patterns, added dependency injections with Guice, and learned how to write better tests. This project not only tested my technical skills but also helped me think more like a systems architect. I understand now why clean architecture, modular code, and dependency injection are so important in building systems that are ready to grow. The overall system and developing it since object oriented one showed me how an entire application can be written. I had struggled with this concept before as many of the programs that I had written were in Python and served a single purpose, not as a single unit. The assignment successfully brought together theory and practice, and I have learned so much.

References

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

GeeksforGeeks. 2023. “Java Dependency Injection (DI) Design Pattern.” GeeksforGeeks. GeeksforGeeks. December 6, 2023. https://www.geeksforgeeks.org/dependency-injection-di-design-pattern/.

“Guice 7.0.0.” 2025. Github.com. 2025. https://github.com/google/guice/wiki/Guice700.

Obregon, Alexander. 2023. “Get Started with Guice: Java Dependency Injection | Medium.” Medium. April 16, 2023. https://medium.com/@AlexanderObregon/introduction-to-guice-a-comprehensive-guide-for-java-developers-489356b6873e.