**Week-10 WriteUp** for

Master of Science

Information and Communication Technology

Krishna Susritha Mylavarapu

University of Denver University College

May 4, 2023

Faculty: Dr. Mike Prasad

Director: Cathie Wilson, M.S

Dean: Michael J. McGuire, MLS

**Write Up**

Throughout this course, I've had the opportunity to apply the concepts I've learned practically. For instance, I developed a Parking System application for the University, incorporating various design patterns, serialization, and multi-thread concepts. This real-world application allowed me to understand the relevance and effectiveness of these concepts. I also learned how to develop a privacy policy and translate UML to Java, further enhancing my practical skills. Additionally, I gained proficiency in using design patterns like Factory, Strategy, Builder, Observer, and Proxy patterns, and learned how to create a client-server architecture using serialization. I have used JDK 1.8 to build, compile, and test my code throughout my application development process.

When I began this assignment, I was uncertain if I would be able to complete it. I initially found the requirements to be quite challenging. However, I found that by taking small steps, such as creating simple classes first and then adding relationships with the new class for each assignment, I was able to make progress. Working on the assignments and receiving Professor feedback helped me submit efficient code. Throughout this course, I had the opportunity to work with Java frameworks and testing frameworks, allowing me to learn and apply various Java concepts.

If I had more time, I would have tried to validate all the inputs received from the user, such as mobile number, license plate, and zip code. I would have even added some security features by implementing the Proxy Design pattern so that people would be authorized first and then given access to some parts of the application. I would have implemented a user-friendly interface for the application.

**UML Diagrams**

Fig 1: Creating Register commands for Car and Customer

In the ParkingOffice class, I've incorporated ParkingService as a new field and created two car and customer registration methods. These methods, when invoked, call the performCommand method in the ParkingService class with the respective command strings. In the ParkingService class, the parkingOffice and commands map are the fields. These fields are initialized in the parameterized constructor of the ParkingService class. When the ParkingOffice invokes the performCommand method in ParkingService, the method splits the Strings from the array of Strings into key-value pairs. It sets the respective fields in the Properties class. Once all the key-value pairs are set, the execute function is invoked in the command class.

The checkParameters method is crucial for validating the Properties attribute in the execute function. This method ensures that all the required attributes are not null. If all the required attributes are present, the method returns true; otherwise, it returns false. This Boolean value is then passed back to the execute function. If it's true, a valid permitId or customerId is provided; if it's false, a null string is returned. The Properties class, in this context, contains all the necessary fields for the Car and Customer.

A diagram of a parking lot

Description automatically generatedFig 2: Developing Strategy design pattern to charge the customer based on the strategy.

I have my strategy classes under the edu.du.ict4315.parking.charges.strategy package. I have created a PricingStrategy interface and implemented multiple classes to execute the methods. I have developed five different pricing strategies to calculate the charges. To make the underutilized parking lots more attractive, I devised a strategy to decrease the parking fee for those lots by 10%. To achieve this, I created a class called LotUsageBasedPricingStrategy. The second strategy I came up with involves decreasing the parking fee by 10% if the car is parked during off-peak hours. I implemented this in the class TimeBasedPricingStrategy.

I have implemented several pricing strategies for the parking lot. Firstly, I have applied differential pricing based on the day of the week. For weekends, I have reduced the parking charges by 10% as the parking lots are less occupied on those days. This functionality has been implemented in the DayBasedPricingStrategy class. Secondly, I have introduced dynamic pricing for special events such as concerts, sports games, or festivals. For these days, I have increased the parking fee by 20% due to the high expected demand and popularity. This strategy has been implemented in the SpecialDayBasedPricingStrategy class. Lastly, I have implemented a strategy based on the type of car being parked. If the car is COMPACT size, a 10% discount is applied. This functionality has been implemented in the CarTypeBasedPricingStrategy class.

A screenshot of a parking system

Description automatically generatedFig 3: Implementing Factory Design Pattern.

I have organized my factory classes under the edu.du.ict4315.parking.charges.factory package. To implement the factory design pattern, I created an interface called ParkingChargeStrategyFactory. This interface contains only the getStrategy method. Another class, ParkingChargeStrategyImpl, has been created to implement the interface. In this class, the method is implemented to return the ParkingChargeStrategy object based on the input in the function parameter.

In the ParkingCalculator class, the getStrategy method is used to return different pricing strategy objects based on the value of the strategy parameter. If the strategy parameter is set to "CarType," the CarTypeBasedPricingStrategy object will be returned. If it is set to "DayBased," the DayBasedPricingStrategy object will be returned. For "LotUsage," the LotUsageBasedPricingStrategy object will be returned. "SpecialDay" will return the SpecialDayBasedPricingStrategy object, and "TimeBased" will return the TimeBasedPricingStrategy object. If an invalid value or null is provided, a null object will be returned.

The getParkingPricing method in the ParkingCalculator class is used to calculate all price changes. It is then called in the ParkingLot and ParkingLotExitEntry classes to retrieve the updated prices. Additionally, the TransactionManager class includes methods to return parking charges based on the license plate number or the customer ID.

A screenshot of a computer

Description automatically generated  
Fig 4: Modifying ParkingLot class to include Observer Pattern.

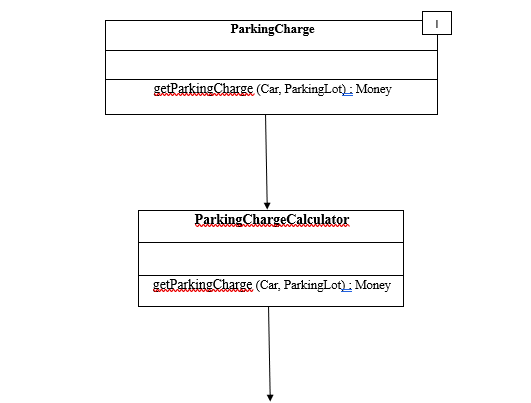
A diagram of a parking system

Description automatically generated  
Fig 5: Implementing Observer Design Pattern.

I have implemented the ParkingObserver class from the ParkingAction interface. The ParkingObserver class includes an eventMap field, which is used to store events when a car enters the ParkingLot and remove them when the car exits the ParkingLot. The overridden method from ParkingAction will be left empty, and the implementation will be added to the subclasses. I have added methods such as addEvent to add the event to the eventMap when a car enters the lot, removeEvent to remove the event from the eventMap when the car exits the lot, and isCarParked to check if the car is present in the eventMap or not.

I have two child classes that extend the ParkingObserver class. One is called EntryParkingObserver, and the other is called ExitParkingObserver. The EntryParkingObserver class only processes events with the eventType "entry," while the ExitParkingObserver processes events with the eventType "exit." If the ParkingLot is an entry-only parking lot, then the parking charges are calculated in the EntryParkingObserver. Otherwise, the events are added to the eventMap, and the parking charges for these cars will be calculated under the ExitParkingObserver.

I have updated the ParkingLot class to incorporate the observer pattern. I added a new field to store the list of observers registered to the ParkingLot. Additionally, I included methods for registering and unregistering observers, a method to notify observers, and a method called entryEvent to trigger the notification process when a car enters the lot. As for the ParkingLotEntryExit class, it extends the ParkingLot class, and I only added the exitEvent method to notify observers when a car exits the lot. Furthermore, I created a ParkingEvent class with the fields car, lot, and eventAction.

A diagram of a parking lot

Description automatically generated  
Fig 6: Represents the Decorator Design pattern in the Parking System application.

I have created an interface called ParkingCharge. Inside this interface, there is a method called getParkingCharge, which returns the charge for parking a vehicle. I have implemented the ParkingChargeCalculator class from the ParkingCharge interface. In the ParkingChargeCalculator class, I have implemented the getParkingCharge method and it returns an empty Money object. This getParkingCharge method takes Car and ParkingLot objects as inputs to calculate the parking charge. Additionally, I have created a class called ParkingChargeCalculatorDecorator, which extends the ParkingChargeCalculator class. In the ParkingChargeCalculatorDecorator class, I have a ParkingChargeCalculator object as a field, and in the getParkingCharges method, I return the parking charges from this field.

I have three child classes that extend the ParkingChargeCalculatorDecorator class. They are FlatRateCalculatorDecorator, CompactCarDiscountDecorator, and LotUsageDiscountDecorator. The FlatRateCalculatorDecorator class is responsible for calculating the parking charges without any discounts being applied. The getParkingCharge method will first check if the ParkingLot instance is an entry-exit lot or an entry-only one. If it is an entry-exit lot, the charges are calculated based on the number of hours the car is parked. If it is an entry-only lot, the charges are calculated based on whether the car will be parked overnight or during the day.

In the CompactCarDiscountDecorator, a 20% discount will be applied if the customer has parked a COMPACT-type car. The getParkingCharge method checks the car type and applies the discount accordingly. In the LotUsageDiscountDecorator, a 10% discount will be applied if the customer has parked the car in a less frequently used parking lot. The getParkingCharge method checks if the lot is frequently used or not and applies the discounts accordingly.

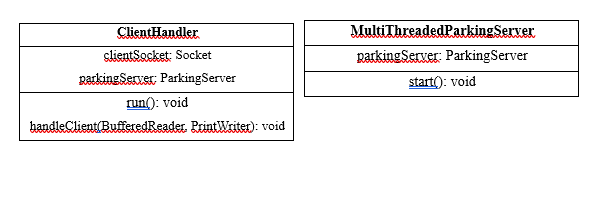
A screenshot of a computer

Description automatically generated  
Fig 7: Implementing Serialization in Parking System application.

The request is located in the package edu.du.ict4315.parking.request. The response class is in edu.du.ict4315.parking.response. The client class is in edu.du.ict4315.parking.client, and the server class is in edu.du.ict4315.parking.server. I have implemented the ParkingRequest class, which allows the construction and sending of request objects to the server. This class has fields for commandName and properties. The commandName field stores the type of command, whether it is a request to register a car or a customer. The properties field contains the list of properties used to register a customer or a car object. Additionally, I have included methods toJsonString and fromJsonString, which are used to convert the request object to a JSON string and to convert a JSON string to a request object.

I have implemented the ParkingResponse class, which serves as a model class for the response object. In the ParkingResponse class, I have defined the fields statusCode and message. The statusCode field holds the status code of the response object, while the message field holds the message of the response object. A statusCode of 200 indicates that the request has been processed successfully, whereas a statusCode of 500 indicates that there may be errors in the request, and the user should rectify the request. Additionally, I have included the methods toJsonString and fromJsonString, which are used to convert the response object to a JSON string and convert a JSON string to a response object, respectively.

The ParkingServer class acts as the server class, where I handle all the requests and send the responses to the clients. It has ParkingOffice and ParkingService as its fields. The methods processRequest and processParkingRequest are also included. The processRequest method converts the JSON request string to a ParkingRequest object. This method then internally calls the processParkingRequest, in which the request is processed, and the objects are registered. Subsequently, the processRequest method receives the ParkingResponse object from the processParkingRequest method and returns the response object as a JSON string.

Fig 8: Implementing Multi-threaded Parking Server Application

I have created the MultiThreadedParkingServer class to handle all client requests and send responses. This class includes a field called ParkingServer. Within the class, I set up a server socket and a client handler using the start method. A new thread is created for the client handler and then started. The MultiThreadedParkingServer class also serves as an application class, so when the application is launched, the multi-threaded server starts running.

I've created the ClientHandler class, which contains Socket and ParkingServer as its fields. It receives input from the console based on the user's commands. If the user enters the "CUSTOMER" command, the code prompts for the customer's name, address line 1, city, state, zip code, and mobile number. If the user enters the "CAR" command, the code requests the customer ID, license plate number, and car type. Entering "exit" exits the Thread. If the user enters anything other than "CUSTOMER" or "CAR," an exception is thrown. Additionally, if any parameters are null, an exception is thrown, and the thread is exited.

I have developed an application class called MultiThreadedParkingClient, which serves as a client in the client-server architecture. In the main method, I initialized a socket and created a PrintWriter object to send data to the server and a BufferedReader object to receive data from the server. If the user inputs "exit," the socket is then closed.

A screenshot of a computer

Description automatically generated  
Fig 9: Implementing annotations using the Guice library.

I wrote the ParkingModule class, which extends the AbstractModule class. I created constructors for the MultiThreadedParkingServer, ParkingServer, ParkingOffice, and ParkingService classes with all the dependent objects. Instead of explicitly initializing them, I used the @Inject annotation to inject all the dependent objects in the classes. For the test classes, I utilized the ApplicationTest class as the base test class and created an Injector object. I used injector objects to create instances of the ParkingOffice and ParkingServer classes. Then, I extended all the test classes from the ApplicationTest class to avoid repeating the same code in all the classes.

**Unit Test cases**

I am using JUnit 4.13.2 to run my unit test cases. I have written unit test cases for all the classes mentioned above, and here are the execution results. My Factory test classes are located in the edu.du.ict4315.parking.charges.factory.test package, Decorator test classes are in the edu.du.ict4315.parking.decorator.test package, Observer test classes are in the edu.du.ict4315.parking.observer.test package, Parking server test class is in the edu.du.ict4315.parking.server.test package and my other test classes are in the edu.du.ict4315.parking.test package.

A screenshot of a computer

Description automatically generated  
Fig 10: Executing Unit test cases from all the packages.