Final Assignment

Parking System Application

for

Master of Science

Information and Communications Technology

Object Oriented Methods & Programming II

ICT 4315-1

Richard Magill

University of Denver University College

June 4, 2021

Faculty: Robert Judd, PhD

Director: Michael Batty, PhD

Dean: Michael J. McGuire, MLS

Parking System Application

ICT 4315-1

Object Oriented Methods & Programming II

Introduction

A Parking System application was created to facilitate common parking activities such as customer registration, vehicle registration, fee calculation, and parking. The application was written in Java using the Netbeans IDE and Junit for unit testing. Some design elements incorporated into the design include:

- Client-Server application
- Data serialization
- Several common design patterns (Factory, Decorator, etc.)
- Multithreading support
- Dependency injection methodology including use of Google Guice

This document details implementation of the key elements, associated unit tests, and recommended improvements and enhancements that should be considered.

Client-Server application

The application is written as a Client-Server application, serializing the data between the Client and Server. In this way the Client and Server are loosely coupled and have improved scalability as compared to a monolithic design (Usman 2002).

The Client contains a list of Commands in the form of a two-dimensional array of strings. My Commands are as follows:

```
public static final String [ ][ ] COMMANDS = new String[ ][ ] {
  {"View Customers", "CUSTOMER"},
  {"Register Vehicle", "CAR", "CarType", "LicensePlate", "Owner"},
  {"Park Car", "PARK", "Permit Id"},
  {"Get Charges", "CHARGES", "Customer"},
};
```

Note that in an earlier iteration of this code instead of a "View Customer" command, I had a "Register Customer" command. However, I decided Customer registration could be implemented as part of Car registration, since the Customer object is passed in with the Car. In order to do this, I implemented some code to check if the Customer is already in the system, and if not, it registers the new Customer with the Car. If an existing Customer was found, it uses the existing Customer object instead. I still found it useful to view the Customer list, so I added a "View Customers" command.

The critical code in the Client is in the *runCommand()* method. Here, I establish a socket connection and a Command object for serialization. Again, in a previous iteration of my code I had created Customer and Car objects for serialization, but there was no need for the Client to be aware of those objects. For abstraction purposes, the Client only needed to know that it was sending a Command and its data (in the form of Map<String,String>) , and once the Server received the Command and data it could create the objects (Customer, Car, etc.) on the Server side.

Some elements of the *runCommand()* are detailed below, as it bears more discussion:

```
//create a commandObject we can serialize to the server
Command commandObject = new Command(command,command,fieldNames);
```

```
            //define a clientOutputStream to the server
            ObjectOutputStream clientOutputStream = new
ObjectOutputStream(link.getOutputStream());


            //write the command to the Server,write the data to the server
            clientOutputStream.writeObject(commandObject);
            clientOutputStream.writeObject(data);


            //create a clientInputStream from the server to get the response
            ArrayList<String> response = new ArrayList<>();
            ObjectInputStream clientInputStream = new ObjectInputStream(link.getInputStream());
            response.add(clientInputStream.readObject().toString());
```

After creating the Command object, an ObjectOutputStream is created, and the Command is serialized to the Server. The related data is also serialized to the Server. An ObjectInputStream was then used to receive the response from the Server.

With respect to the Server, I get the ObjectInputStream provided by the Client and deserialize it with the *readObject()* method. Using the Command "name" attribute, I then know which method to use to process the Command. A response back to the server is provided by the method as well:

```
                try {
                   //first deserialize and read the command
                   Object object = serverInputStream.readObject();
                   Command command = (Command)object;


                   //next read the command arguments
                   Map<String,String> data = (Map<String,String>)serverInputStream.readObject();
```

```
                    //switch on the command
                    switch(command.name()){
                       case "CUSTOMER":
                          serverOutputStream.writeObject("Registered customers = " +
parkingOffice.getCustomers());
                          break;
                       case "CAR":
                          serverOutputStream.writeObject(processCarCommand(data));
                          break;
                       case "PARK":
                          serverOutputStream.writeObject(processParkCommand(data));
                          break;
                       case "CHARGES":
                          serverOutputStream.writeObject(processChargesCommand(data));
                       default:
                    }
```

The following *processCarCommand* illustrates unpacking the data, creating the Car object, and the previously-discussed Customer registration method within the Car registration process:

```
                private String processCarCommand(Map<String,String> data) throws Exception{
                   //first unpack the data on the car, CarType, LicensePlate, Owner
                   String[] dataArray = data.values().toArray(new String[3]);
                   CarType carType = CarType.valueOf(dataArray[0]);
                   String licensePlate = dataArray[1];
                   //then see if customer exists, if not register them
                   Customer customer = parkingOffice.getCustomerByCustomerName(dataArray[2]);
                   if (customer == null){
```

```
                customer = new Customer(dataArray[2]);
                parkingOffice.register(customer);
            }
            //register the car and write response back to client
            Car car = new Car(carType, licensePlate, customer);
            parkingOffice.register(car);
            return "Registered vehicles = " + parkingOffice.getPermits();
        }
```

While this serialization approach is functional for prototyping, it should be noted that Java serialization in this fashion is out-of-favor due to security concerns (Krill 2019). I've started refactoring and have implemented classes using this JSON approach on the Client side, but ran out of time to complete the implementation on the Server side. Completion of that revision will be a necessary improvement to the application. The partially complete JSON packages and classes are included in the code.

The ClientJSON serializes in a manner as follows:

```
            //write the command to the Server,write the data to the server
            ObjectMapper objectMapper = new ObjectMapper();
            ObjectNode node = objectMapper.createObjectNode();
            node.put("action", "customer");
            node.putPOJO("details", command);
            node.putPOJO("data", data);
            String json = objectMapper.writeValueAsString(node);
```

This Client creates JSON:

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ MagillParkingClientServerJSON ---
JSON to Server:
{"action":"customer","details":"CAR","data":{"CarType":"SUV","LicensePlate":"123","Owner":"bob"}}
```

A ParkingGui class is also included. Figures 1 and 2 below illustrate the ParkingGui and car registration:

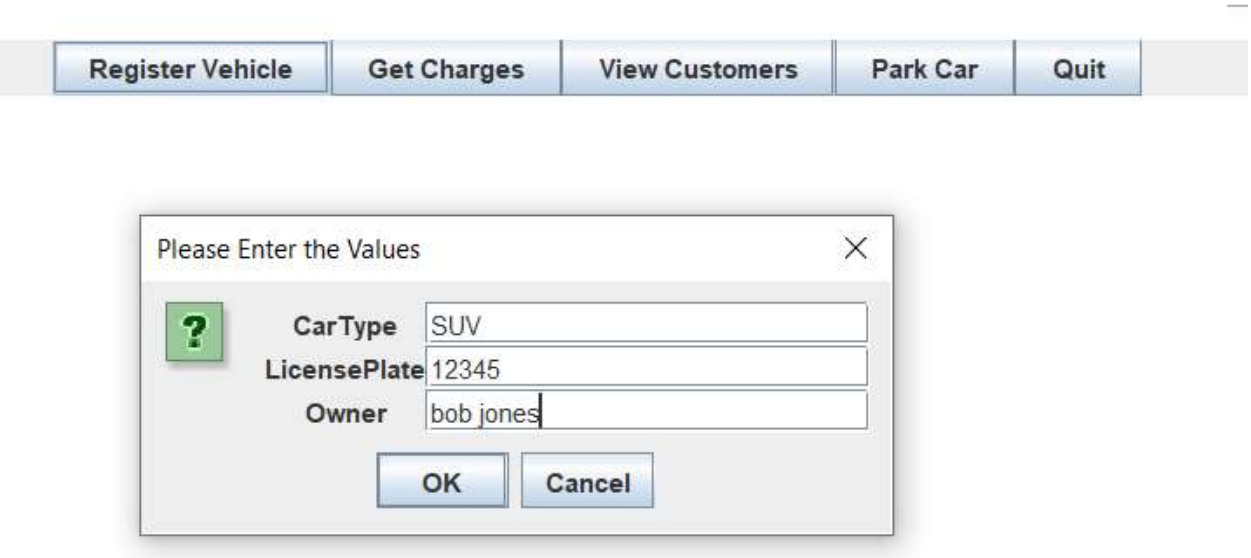

Figure 1: Registering a Vehicle

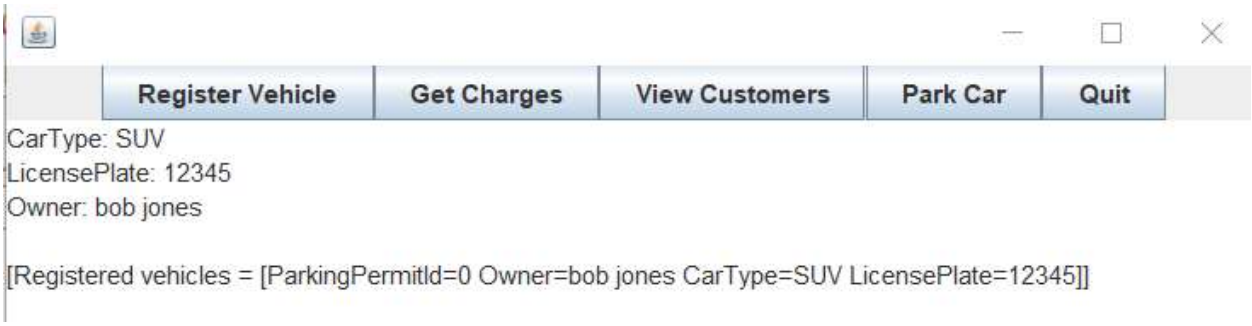

Figure 2: Post registration, showing response from Server

For more notes on application usage, please see *Appendix A -ReadMe*, which details inputs, function, and expected behavior for each command.

The Server interacts with the classes that comprise the business logic of the application, such as Car, Customer, ParkingOffice, etc. All parking data(Customers, Cars, Transactions) is currently stored in memory in the ParkingOffice, in the form of List<T>. However, another necessary improvement to the application is to implement some persisted data storage. I recommend a RDBMS such as MS SQL or MySQL to persist the data between sessions. While data storage in a file would be possible, with any significant usage we would need to use a database.

In summary, the ParkingGui uses the Client to serialize the Command and its data to the Server. The Server deserializes the Command and the data. Depending on the Command name, the Server creates the necessary objects and processes the Command with the data provided, and then writes a response back to the Client. For example, the "Registered Vehicles = …" information above was serialized back to the Client from the Server.

A package diagram for the application is shown in Figure 3 below:

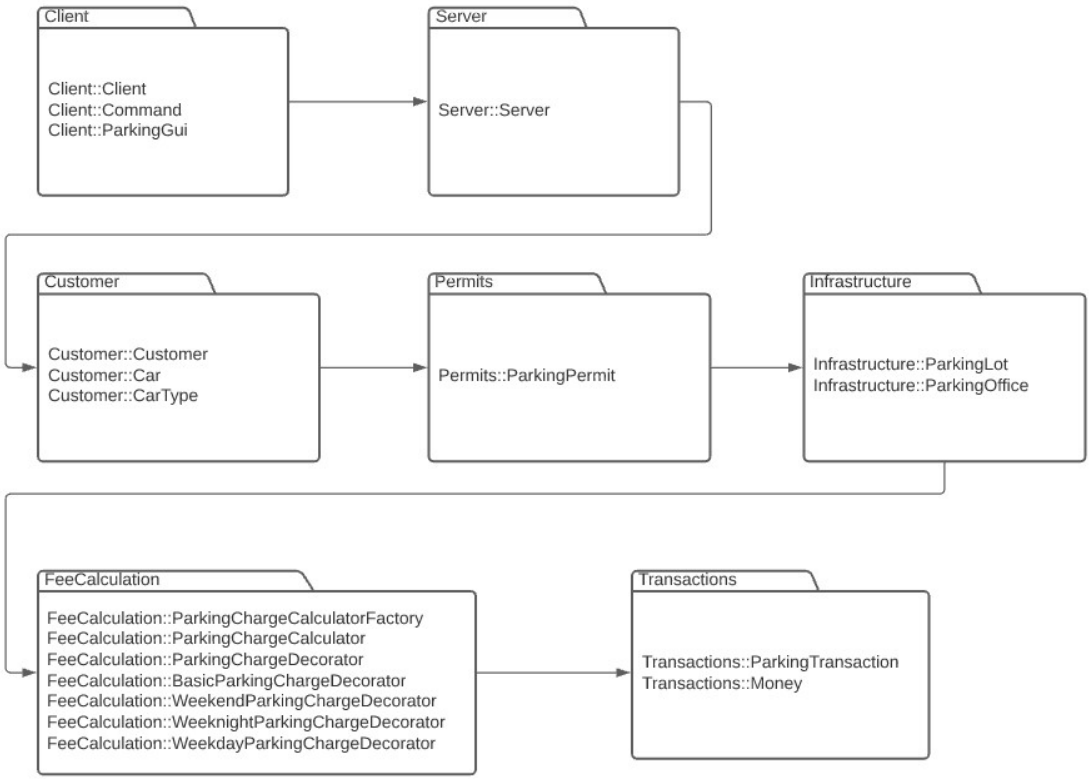

Figure 3: Parking System Application UML Package Diagram

Design patterns

Design patterns were critical in implementation of this application. The following details the evolution of that usage in my application.

Command Pattern

One of the first patterns introduced was the Command Pattern. A "Command" interface was created that provided the abstract method definitions for the Command classes. The RegisterCustomerCommand class and the RegisterCarCommand class implemented the Command interface, and both classes included a method called "execute" that registers either the Customer or the

Car. The purpose was to encapsulate all information needed to perform the command in the concrete Command objects.

In my Client-Server application the Command is a single concrete class, and the instantiated object is serialized to the Server. So, in my final implementation there are no RegisterCustomerCommand or RegisterCarCommand classes.

Strategy Pattern

The next implemented pattern was the Strategy Pattern. A FeeStrategy pattern was created that allowed the ParkingLot to get the appropriate FeeStrategy based on *parkDate* and *CarType*. Being able to select the algorithm at run-time means that a single complex fee calculation method was not required.

Factory Pattern

The Factory Pattern was first employed to increase the level of abstraction between the ParkingLot and the FeeStrategy. Using the Factory Pattern provided an abstract selection of the FeeStrategy. ParkingLot didn't need to know nothing about how to select the FeeStrategy or even which FeeStrategy was returned. The only responsibility for the ParkingLot class is to provide a parking date and CarType to the Factory, which then returns the correct FeeStrategy. This FeeStrategy approach was subsequently replaced with a Decorator pattern.

Observer Pattern

The Observer pattern (also called Pub-Sub) introduced the ability for the ParkingLot class (the Subject) to notify an object (the ParkingObserver) upon entry or exit from the lot. The Observer tracks the state of the Subject while remaining only loosely coupled to the Subject. The main objective here was to introduce a way to track ParkingEvents as they occur. With the implementation of my Client-

Server class, the Observer pattern was no longer required since the notifications of a *park()* event are now provided to the Server and downstream application by the loosely coupled Client.

Decorator Pattern

The FeeStrategy and FeeStrategyFactory patterns were replaced with a ParkingChargeCalculatorFactory and a Decorator pattern. The main objective here was to abstract complexity out of the ParkingLot class. The ParkingLot doesn't know how the parkingChargeCalculator is decorated, it just gives parameters to the Factory and gets back the correct calculator. This Decorator Pattern and the related Factory Pattern remain in the current design, as detailed in Figure 4.

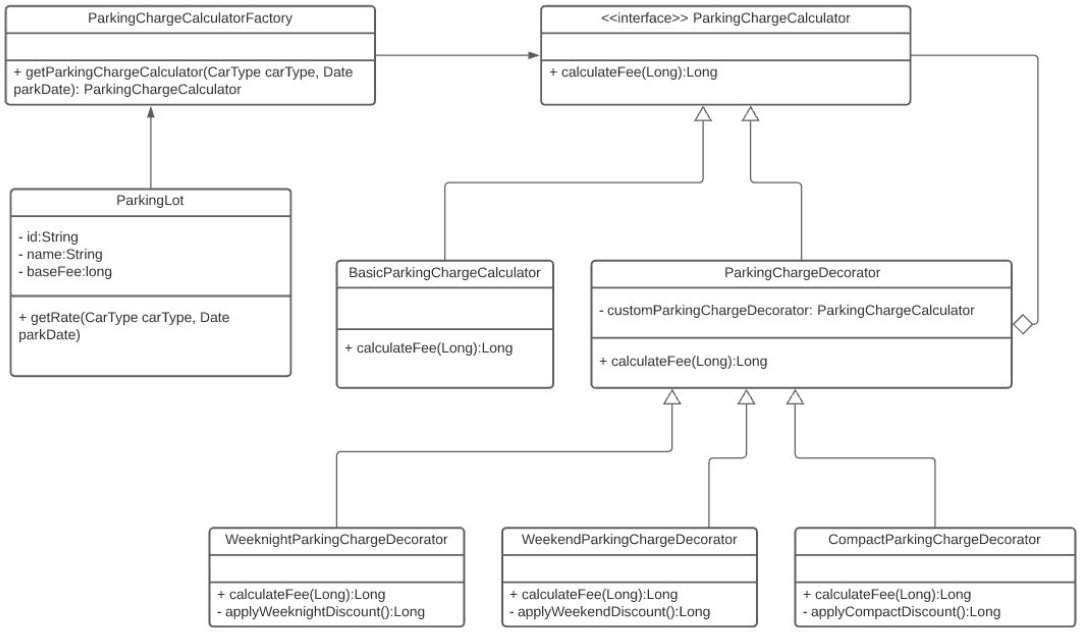

Figure 4: Factory and Decorator Pattern, Fee Calculation

UML

The overall application architecture is illustrated in Figures 5 and 6 below, omitting the Fee classes previously illustrated in Figure 4.

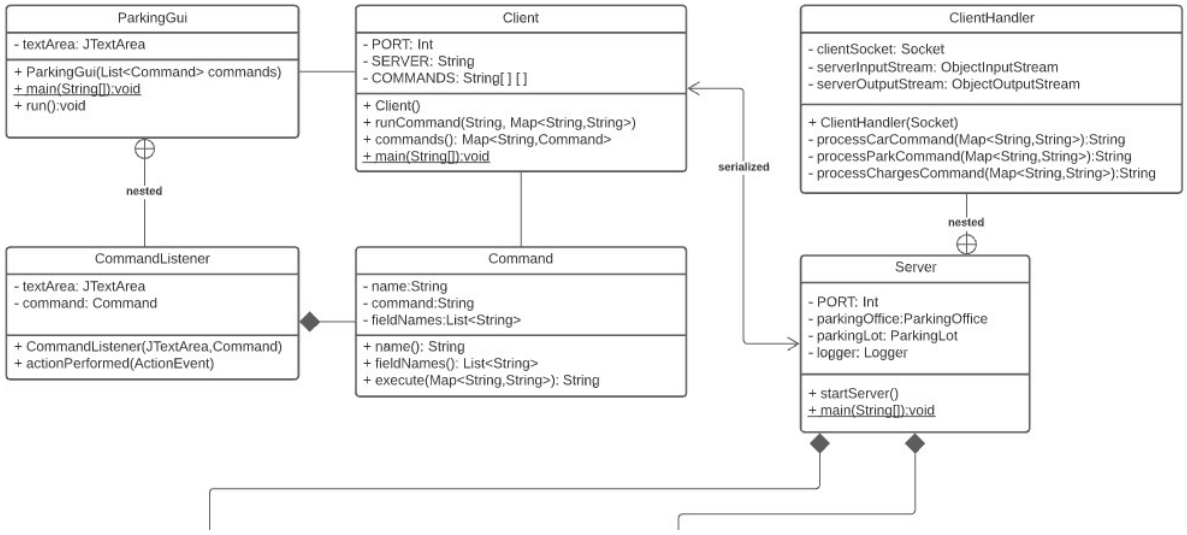

Figure 5: Client and Server UML

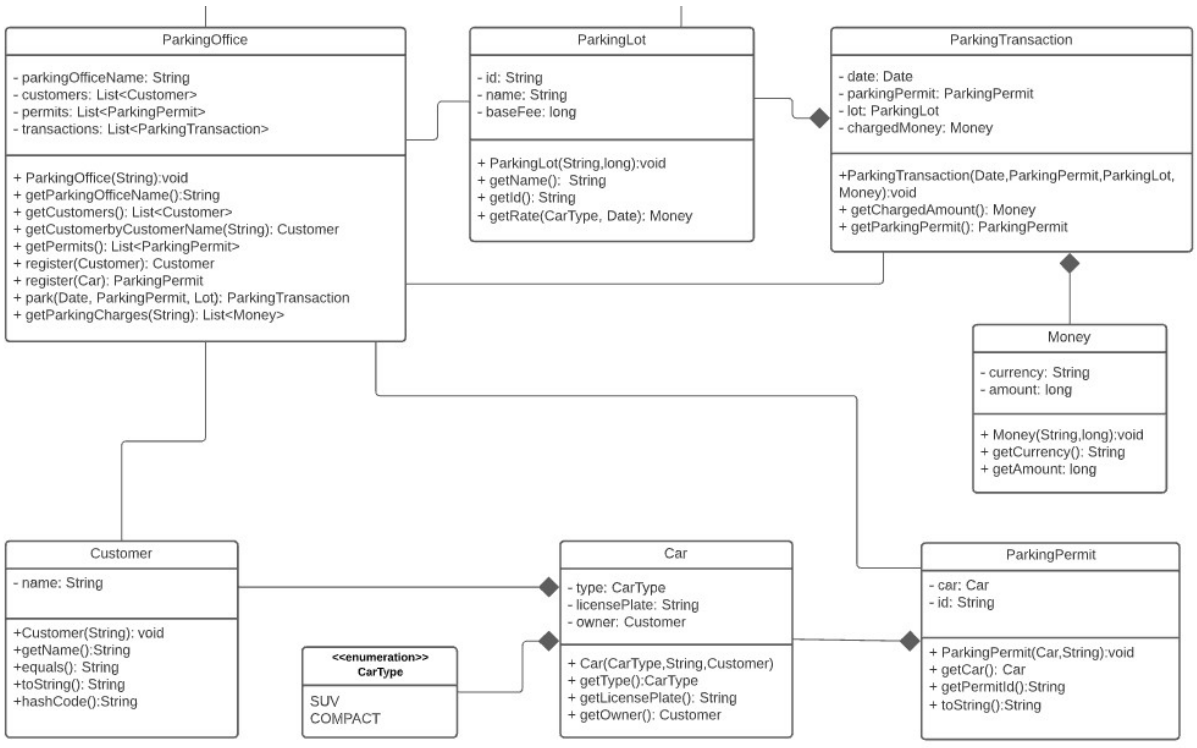

Figure 6: Parking System UML

Multithreading

Multithreading is necessary to make the Parking System application capable of handling multiple Clients, as we would expect if we were to have multiple users of our system. Multithreading facilitates more responsive code and less user wait time. It also ensures the threads process data independently from one another (Wong and Oaks 2004).

In order to modify the Server class for multithreaded capability, I created a new ClientHandler class that implements the Runnable interface. The Runnable interface contains a single method, *run(),* which was then implemented concretely within my ClientHandler to read the object passed from the ClientInputStream.

Implementation of the ClientHandler class is as follows, note here I've replaced some of the code with ellipses for brevity, as I will do throughout the paper:

```
// ClientHandler class implements runnable for multithreading
  private static class ClientHandler implements Runnable {
     private final Socket clientSocket;
     // Constructor
     public ClientHandler(Socket socket)
     {
        this.clientSocket = socket;
     }
     public void run()
     {
         [ . . . ]
     }
  }
```

A new Thread object was created within the Main method, and the *start()* method was then invoked on the newly created Thread object:

```
public void startServer() throws IOException, ClassNotFoundException {
  logger.info("Starting server: " + InetAddress.getLocalHost().getHostAddress());
  try (ServerSocket serverSocket = new ServerSocket(PORT)) {
    serverSocket.setReuseAddress(true);
    while (true) {
      Socket client = serverSocket.accept();
      // Displaying that new client is connected to server
      System.out.println("New client connected"
                         + client.getInetAddress().getHostAddress());
      // create a new thread object
      ClientHandler clientSock = new ClientHandler(client);
      // This thread will handle the client separately
      new Thread(clientSock).start();
    }
  }
}
```

Finally, it was important to ensure access to shared data was fully protected. In particular, I needed to ensure that the Parking Office was synchronized, which I did with synchronized code blocks as follows:

```
private String processCarCommand(Map<String,String> data) throws Exception{
    List<ParkingPermit> permits;
    synchronized(parkingOffice)
    {
      [. . .]
```

```
            Customer customer = parkingOffice.getCustomerByCustomerName(dataArray[2]);
            [. . .]
        }
```

Dependency Injection

Inversion of Control is a software design principle which seeks to transfer control of objects to a container or framework (Alvarez 2017). We essentially enable the framework to create objects, in order to decouple the execution for the task from its implementation. The general concept of IOC can be achieved through a number of mechanisms (Strategy Pattern, Factory Pattern, etc.) but one method is Dependency Injection. Dependency Injection is a method by which an object's dependencies are provided to the object.

One simple way we do this on a frequent basic is by simply passing the dependencies into the object constructor. Google Guice is a Java-based dependency injection framework that provides a way to graph dependencies and remove hardcoding from the code (Bhatia 2018). One key advantage from Guice is the way in which you can override the constructor for unit testing, greatly improving the ability to create independent mock objects for testing.

Since my code originally included hard-coded ParkingOffice and ParkingLot references, I used Google Guice to inject those dependencies in the Server class. After adding the dependency on Guice to the POM file, I was ready to implement the framework. Using the ParkingLot (for example), I annotated the constructor with @Inject. Next, I created a ParkingLotModule for injection, creating the data for a default Parking Lot:

```
public class ParkingLotModule extends AbstractModule {
    @Override
```

```
    protected void configure() {
        bind(String.class)
         .toInstance("Main Street Lot");
        bind(Long.class)
          .toInstance(20L);
    }
}
```

Next, I simply inject this in Server Main method with Guice, creating the ParkingLot:

```
public static void main(String[] args) throws Exception {

    Injector injectorParkingLot = Guice.createInjector(new ParkingLotModule());

    parkingLot = injectorParkingLot.getInstance(ParkingLot.class);
```

The system will now respond exactly as if I had used the traditional construction method with the “new” keyword. However, note my improved ability to override the default module and create mocks for unit testing:

```
    @Before
    public void createFakeLot(){
        com.google.inject.Module testModule = Modules.override(new ParkingLotModule())
            .with(new AbstractModule(){
            @Override
            protected void configure() {
                bind(String.class).toInstance("Fake Parking Lot");
                bind(Long.class).toInstance(20L);
            }
        });
        Injector injector = Guice.createInjector(testModule);
```

```
            lot = injector.getInstance(ParkingLot.class);
        }
```

Unit testing

Unit Testing is vitally important to ensure that code meets quality standards prior to deployment. Ideally, all classes within the project should have relevant methods tested. While much of the code is properly tested, there are still improvements needed to the test set. The status of unit tests at this time are detailed below:

Classes with sufficient testing:

- Infrastructure::ParkingLot: 100% coverage, all methods tested
- Infrastructure::ParkingOffice: 100% coverage, all methods tested
- Fee::[all classes]: 100% coverage, all methods tested
    - Tested within the ParkingChargeCalculatorFactoryTest
- Customer:: Car: has only getters, setters, and toString(), no test required
- Customer:: Customer: has only getters, setters, and toString(), no test required
- Customer:: CarType: enumeration, no test required
- Permit::ParkingPermit: has only getters, setters, and toString(), no test required
- Transaction::Money: has only getters, setters, and toString(), no test required
- Transaction::ParkingTransaction: has only getters, setters, no test required

Classes that still require unit tests:

- Client::
    - I ran out of time to implement tests for the Client, Command, and ParkingGui classes.

    - Given that *runCommand()* is protected, the best way to test this would be to include a test class in the Client package, which can then test protected or package-private methods.
    - Additionally, since Client package elements are currently dependent on a socket connection to a running server, a mock server will have to be incorporated in the unit test.
- Server::
    - Again, the Server class and its nested ClientHandler class will need to be unit tested.
    - Testing the protected methods in the class (performCarCommand, etc.) will also require a test class in the Server package.

Running the TestRunner class shows that all current tests pass unit testing.

Conclusion

This application incorporates design patterns including Factory, Decorator, and Client-Server to create a robust application framework. Serialization of the data has been implemented, using Java object serialization and also via a partially-complete refactoring for JSON serialization. The Server is set up with multi-threading, including protection for the synchronized ParkingOffice and ParkingLot objects. Dependency injection for the ParkingOffice and ParkingLot objects has been implemented using Google Guice, including within the relevant unit tests. Unit testing, while not 100% complete at this time, has been widely implemented.

Next steps to further advance this application should include:

- Addition of data persistence via an RDBMS

- Improvements to the ParkingGui
- More thorough unit testing
- Completion of the JSON serialization refactoring

While more work needs to be done, the application is soundly constructed and incorporates strong object-oriented design principals.

Appendix A – ReadMe

- To run, first start Server, then ParkingGui. The JSON-oriented classes aren't 100% functional yet, so they should not be started.
- To "Register Vehicle", CarType options of SUV or COMPACT are the only options that will correctly Register the car. Changing the CarType input to a dropdown with only those two options would be a ParkingGui improvement that should be made soon.
- "Customer Registration" occurs automatically with Vehicle Registration, as noted earlier.
- Registering a car with a duplicate license plate will retun null to the client and throw an exception:

```
Mon May 31 14:45:19 PDT 2021 SEVERE  (Server.Server$ClientHandler run) null
java.lang.Exception: license plate already registered
```

- To "Park Car", the only required input is the integer corresponding to the Parking Permit Id. So in the example shown above to park that vehicle, just enter 0. The "park time" parking is equal to the time of submission, so no "park time" user entry is required. Also, by convention this lot doesn't do overnight parking. You park and pay for a day, so charges are based on that and I am not currently tracking time of exit. That is probably another improvement that should be made as well. "Park Car" will return the charges from the Server, as in "[Parking Transaction, Id = 0, $20]"
- "View Customers" provides a list of all registered customers.
- "Get Charges" requires just a user name, and then the Server will respond back with a list of all such charges for that user. Another improvement here would be to add more detail to that response, such as license plate and park time.

References

Saleem, Usman 2002. "A Pattern/Framework for Client/Server programming in Java". developer.com, June 10, 2002. https://www.developer.com/java/enterprise-java/a-pattern-framework-for-client-server-programming-in-java/

Krill, Paul 2019. "Oracle plans to dump risky Java serialization". Infoworld.com, May 24, 2018. https://www.infoworld.com/article/3275924/oracle-plans-to-dump-risky-java-serialization.html

Wong, Henry; Oaks, Scott. 2004. *Java Threads, 3rd Edition.* O'Reilly Media, Inc

Alvarez, Francisco 2017. "IOC vs. DI". Dzone.com, Dec. 27, 2017. https://dzone.com/articles/ioc-vs-di

Bhatia, Sankalp 2018. "A hands-on session with Google Guice". freecodecamp.org, Oct. 30, 2018. https://www.freecodecamp.org/news/a-hands-on-session-with-google-guice-5f25ce588774/