
System Design Document

EECS3311: Parking Booking System

Lian Attily (218464016)

Document ID	SYSTEM DESIGN-v0.1
Version Number	1.0
Issue Date	April 24th, 2021
Classification	Public

1.1 Methodology, Tools, and Approach

- Coded in Java (**Java 1.8**)
- Automatically built and tested with **Gradle** . Dependencies include:
 - Junit 5
 - JavaFX (OpenJFX)
 - Jacoco
- GUI is coded using the **JavaFX 15** framework and designed with **SceneBuilder** (styled with CSS)

Please refer to the user manual document in the documentation directory for a full, step-by-step guide on using the software.

Demo video: <https://youtu.be/8WfZxgUr7NI>

2 Detailed Design

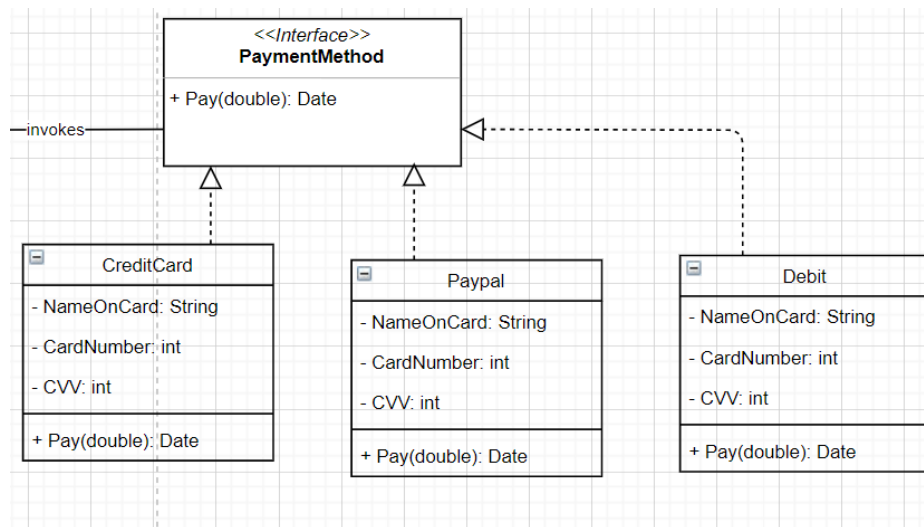
The design of this project favors **composition** over inheritance. For instance, composition allows the system administrator to add and remove parking enforcement officers. Another example is how the officers add and remove Parking Spots, and the system administrator can add and remove Officer objects.

Requirement 3.1 states that the system should provide different login and user options depending on the user type (customer, system administrator, or Toronto Parking Authority). Thus, the **Composite design pattern** is used for the different users using the system. The interface methods can be implemented by any of the three users. For instance, the system administrator class can implement the User interface and override the LogIn() method in order to log in with the master ID and password only. Similarly, the Customer class can override the CreateAccount() and LogIn() methods to create the customer's account and verify the customer's information, respectively. The Toronto Parking Authority class would override the LogIn() method to display the required verification information for a parking enforcement officer. The initial design and implementation use the composite design pattern to implement the different *User* classes as they essentially share some of the same methods.

In the initial design, the **Observer design pattern** was used to design the Parking Spot class (Subject) that contains the parking spot information. The observers of this class are the classes that implement the User interface. The Parking Spot class has getter methods for its attributes and combines all its data and displays them when calling the Display() method. The calculatePayment() method returns the payment amount when it is called any time the customer requests to view their payment amount through the GUI. When data of the parking spot changes (such as spot availability or payment status), the observer classes are notified. For instance, the Parking Spot class (subject) will notify the parking enforcement officer and

the customer (observers) when a parking spot booking expires. During the implementation, however, the GUI classes acted as the subjects rather than the Parking Spot class. The User subclasses, however, remained as observers as they fed the changes from the subject and updated the databases accordingly.

The Payment class is also designed through the **Observer pattern**. The observer, the system administrator, is notified whenever a successful payment for a booked parking spot is made through the notify() method. The Parking Spot class is also an observer of the Payment class and is notified of the payment date and time to begin the countdown to expiration time. This encapsulates REQ 4.6 (specifically, 4.6.2 and 4.6.3-REQ-5). This pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



To implement REQ 4.6 (Payment) the system must maintain the different payment mechanisms with different processes, although they all perform one action (make the payment). Through the **Strategy design pattern**, a **PaymentMethod** interface is created, and the different methods are encapsulated into their own set of classes. This pattern allows the system to define a payment algorithm, encapsulate it, and make it interchangeable. Strategy lets the algorithm vary independently from clients that use it. In both the initial design and implementation, the strategy pattern was used to encapsulate the action of performing a *payment* through the different payment strategies (PayPal, Debit, and Credit Card).

During the initial design of the system, the City of Toronto's parking spots information was to be maintained through the Singleton design pattern that restricts the instantiation of the **CityParkingSpots** class to *one* instance. Similarly, the **Account** class is implemented through the Singleton design pattern. That is, the **Customer** and **Officer** classes can only create one instance of an **Account**. During the implementation, however, the accounts and parking spots are created and removed dynamically, maintained through a database instead, and the Singleton design pattern was not used.

Lastly, the **Visitor design pattern** was used in order to allow the parking enforcement officer to “visit” and view a Customer’s booking details. This pattern helps capture 4.7.3-REQ-1: that only authorized users may view a Customer’s bookings.

The main difference between the initial design of the project and the implementation was how the involvement of the various GUI controller classes altered the design of the software. For instance, in the initial design, the GUI components were incorporated with the object classes, while during the implementation they are completely separated. There are controller classes for each GUI window, which heavily rely on composition of the various object User subclasses.

Classes and Interfaces:

- **User** <<interface>> is implemented by:
 - **Customer:** This class will encapsulate all of the operations that a customer can perform
 - **SystemAdmin:** This class will encapsulate all of the operations that the system administrator can perform.
 - **Officer:** This class will encapsulate all of the operations that the Toronto Parking Authority personnel (parking enforcement officers) can perform
- **PaymentMethod** <<interface>> is implemented by:
 - **CreditCard:** This class will perform and authorize payment by credit card.
 - **Debit:** This class will perform and authorize payment by debit card
 - **Paypal:** This class will perform and authorize payment through paypal.

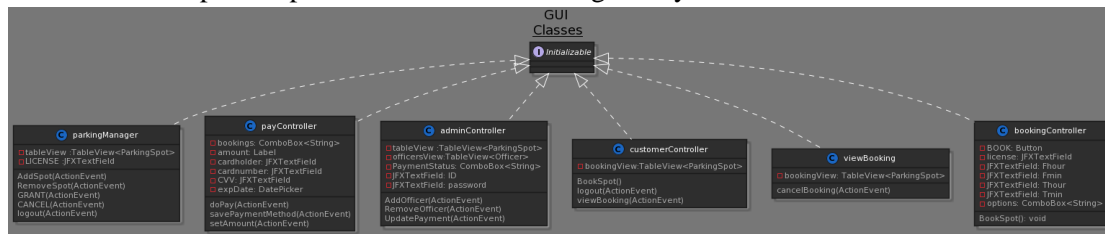
Invoked by:

- **Payment:** This class is responsible for making payments (according to the payment type specified by Customer) and notifying SystemAdmin when a ParkingSpot has been paid for.
- **ParkingRate:** This class will keep track of the hourly parking rates. It will specify a dollar amount for each hour.
- **ParkingSpot:** This class will encapsulate a parking spot and all of its attributes. The Customer will book a ParkingSpot.

GUI Classes:

- **Application** extended by:
 - **Main:** Overrides the start() method which acts as an entry point for the application
- **Initializable**<<Interface>> implemented by:
 - **MainController:** This is the “welcome” window, from where the user can continue as a system administrator, bylaw officer, or a regular user.

- **ParkingManager:** Encapsulates the Parking Authority Officers' window; performing functions like adding and removing parking spaces, as well as granting and cancelling booking requests.
- **adminController:** This class implements all of the GUI functionalities of a system administrator, such as adding/removing Parking Authority Officers, and updating a customer's payment status to PAID.
- **customerController:** This class encapsulates the regular user's main view, showing all of the available spots, and the buttons to book a spot, pay for a spot, or view all bookings.
- **bookingController:** This class implements the booking a spot GUI functionality, prompting the user to pick the desired booking spot, license plate number, start and end times, as well as the booking date. If the customer has 3 or less bookings, booking a spot will stamp the
- **payController:** This class implements the Payment GUI feature, allowing the customer to pick a booked spot, showcasing the amount due, entering their payment information, and either saving the payment information for future use or paying immediately.
- **viewBookings:** This class allows the user to view all of their booked spots, pick a spot and cancel the booking if they wish.



As for right now:

- Master ID: **MASTER**
- Master Password: **MASTER**
- Officer ID & Password: **demo**
- Customer email: **demo@demo.com**
- Customer password: **demo**

Below is a detailed list of how the class diagrams and relationships between classes will encapsulate each of the specifications listed in *Section 4: System Features*. Please refer to the attached PNG for the class diagrams details.

- REQ 4.1: The SystemAdmin class contains a list of officers and the unique ID assigned to them (4.1.3-REQ-1), the system can easily iterate through the list and verify whether an officer exists before adding or removing them from the system and

assign a unique ID to each added officer (4.1.3-REQ-2 and 4.1.3-REQ-3 and 4.1.3-REQ-4).

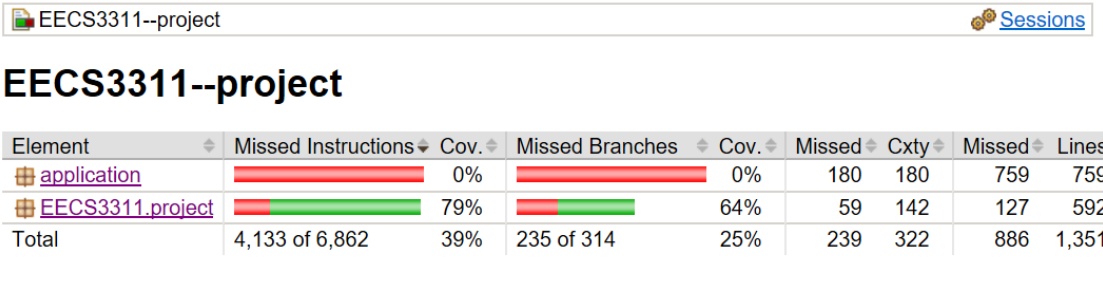
- REQ 4.2: Using the composite pattern, the Customer class implements the User interface and overrides the CreateAccount() method. The GUI controller class (subject) prompts the user to enter the required information, and the Customer class is notified, thereby creating an account, and adding the customer's information to the database (CustomerDatabase.txt) for the future.
- REQ 4.3: The Customer class also overrides the LogIn() method to prompt the user's login information and verify the login information accordingly by checking their existence in the database (CustomerDatabase.txt).
- REQ 4.4: The Officer class adds and removes parking spots to the database (ParkingsDatabase.txt) by their postal code, along with their rate per hour. The user will then select a parking spot and enter how long they are booking it for. The BookSpot() method will return true and assign a unique booking ID if the selected spot is free, or it will return false and display an error message if the selected spot is unavailable (4.4.3-REQ-3). The booking ID is a unique string that is associated with a booked ParkingSpot (4.4.3-REQ-6) in the "BookingsDatabase.txt". The multiplicity of the composition relation between the customer and parkingSpot shows that the customer may have up to 3 booked parking spots only (4.4.3-REQ-5).
- REQ 4.5: After successfully logging in, the customer can click a "Cancel" button, which will prompt the CancelBooking() method. This method will ask the user to enter the ID of the booking they wish to cancel (4.5.3-REQ-2), checks the ParkingSpot's expiration time, and returns true if the booking has not expired (4.5.3-REQ-3).
- REQ 4.6: In the GUI, the Pay button will invoke the pay() method which will display the bookings with PaymentStatus = unpaid. The amount to pay can be obtained by calling the appropriate ParkingSpot class's CalculatePayment() method. It will also prompt the user to enter the payment method they wish to pay with and proceed with the payment. The payment interface is implemented by many different payment strategies. The different payment methods override the pay() method in order to authenticate the customer's payment information accordingly (4.6.3-REQ-3 and 4.6.3-REQ-7). The pay() method returns a date and time that is automatically updated as the ParkingSpot paymentTime, thereby starting the countdown until the booking expires (4.6.3-REQ-4 and 4.6.3-REQ-5).
- REQ 4.7: Because ParkingSpot was designed using the Observer pattern and the Customer is an "observer", the ViewBookings button will invoke the viewBookings()

method, which will display the details of the customer's booked parking spots by easily iterating through the bookings list and calling the ParkingSpot's Display() method to display the parking spot's details to the customer. Similarly, the parking enforcement officer, as a visitor of the Customer class, may also call the customer's ViewBookings() method to view their booking details (4.7.3-REQ-1). Furthermore, the ParkingSpot will notify the observer, the customer, when their booking has expired (4.7.3-REQ-3).

- REQ 4.8: The officer can add space, remove space, cancel/grant requests easily through methods invoked by the buttons for each action. When an officer wants to perform an operation on a ParkingSpot, they will click on the ParkingSpot from the TableView where they can: remove space, cancel/grant requests - that will call appropriate methods. They can also enter a Postal code and rate/hour and add a Parking Spot to the system. Before performing an operation, they will iterate through the ParkingSpots list to verify if it is vacant (4.8.3-REQ-2 and 4.8.3-REQ-4).
- REQ 4.9: Once the user makes a payment, the PaymentGUI Controller subject will notify the System Administrator observer by returning the Customer object. Initially, they can change the PaymentStatus of the parking spot from unpaid to paid by entering the customer's information and manually changing the payment status to "paid". However, during the implementation it was found to be more efficient to pick a booked ParkingSpot and change its Payment Status from PENDING to PAID. If the customer is found in the database, the system administrator can "visit" the customer and view their bookings to verify the customer's occupancy of the parking space (4.9.3-REQ-2 and 4.9.3-REQ-3). In order to satisfy 4.9.3-REQ-4, the system administrator will be notified to change the payment status AFTER the customer successfully pays for the booking.

<<enumeration>> <u>PaymentStatus</u>	
	Paid Unpaid Pending

3. Code Testing



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines
application	<div><div></div></div>	0%	<div><div></div></div>	0%	180	180	759	759
EECS3311.project	<div><div></div></div>	79%	<div><div></div></div>	64%	59	142	127	592
Total	4,133 of 6,862	39%	235 of 314	25%	239	322	886	1,351

Figure 1: Jacoco Code Coverage Report

None of the GUI classes have been tested, but the non-GUI classes (EECS3311.Project package) shows coverage of ~80%. After running the unit tests from the Gradle tasks, refresh the project folder. The Jacoco Testing Coverage can be found through the build directory → JacocoHtml → Index as shown in the figure below:

