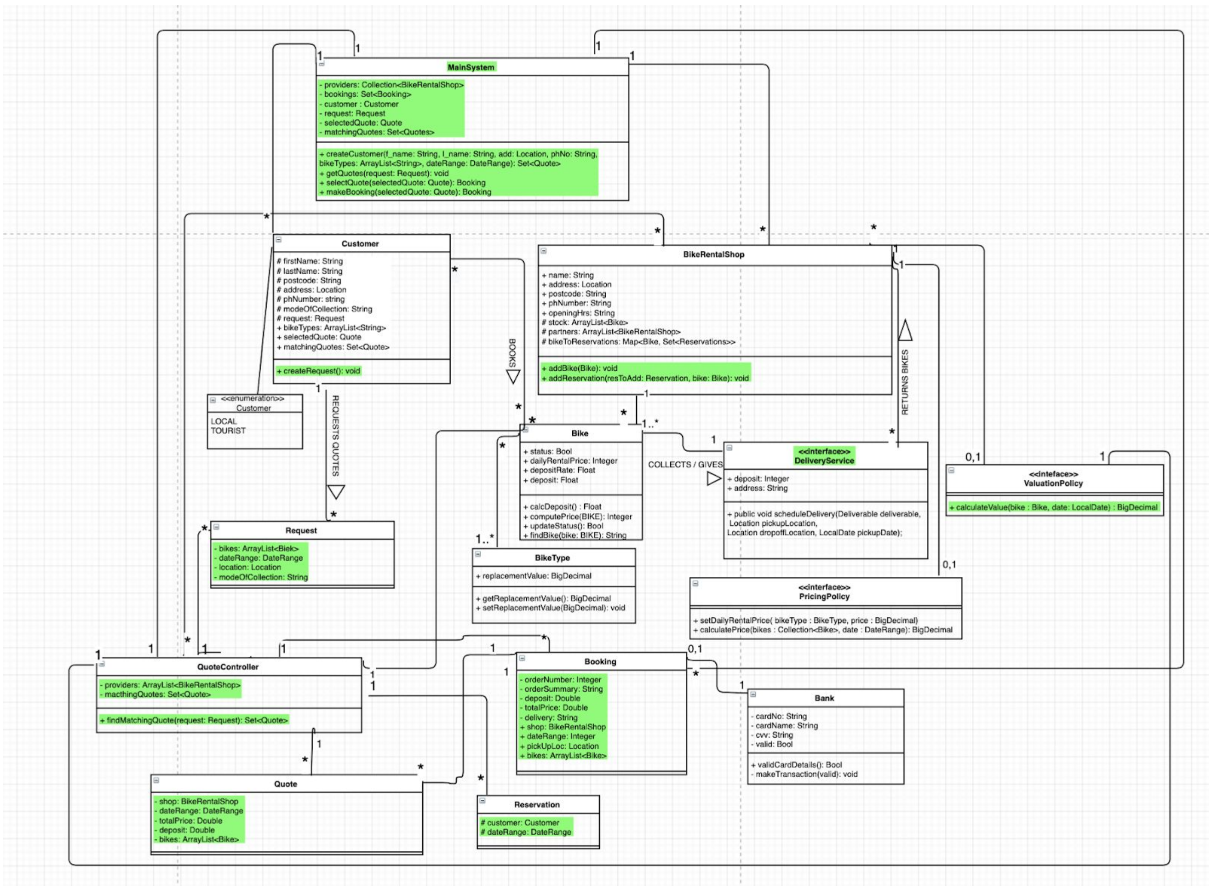
Inf2C - Software Engineering 2019-20

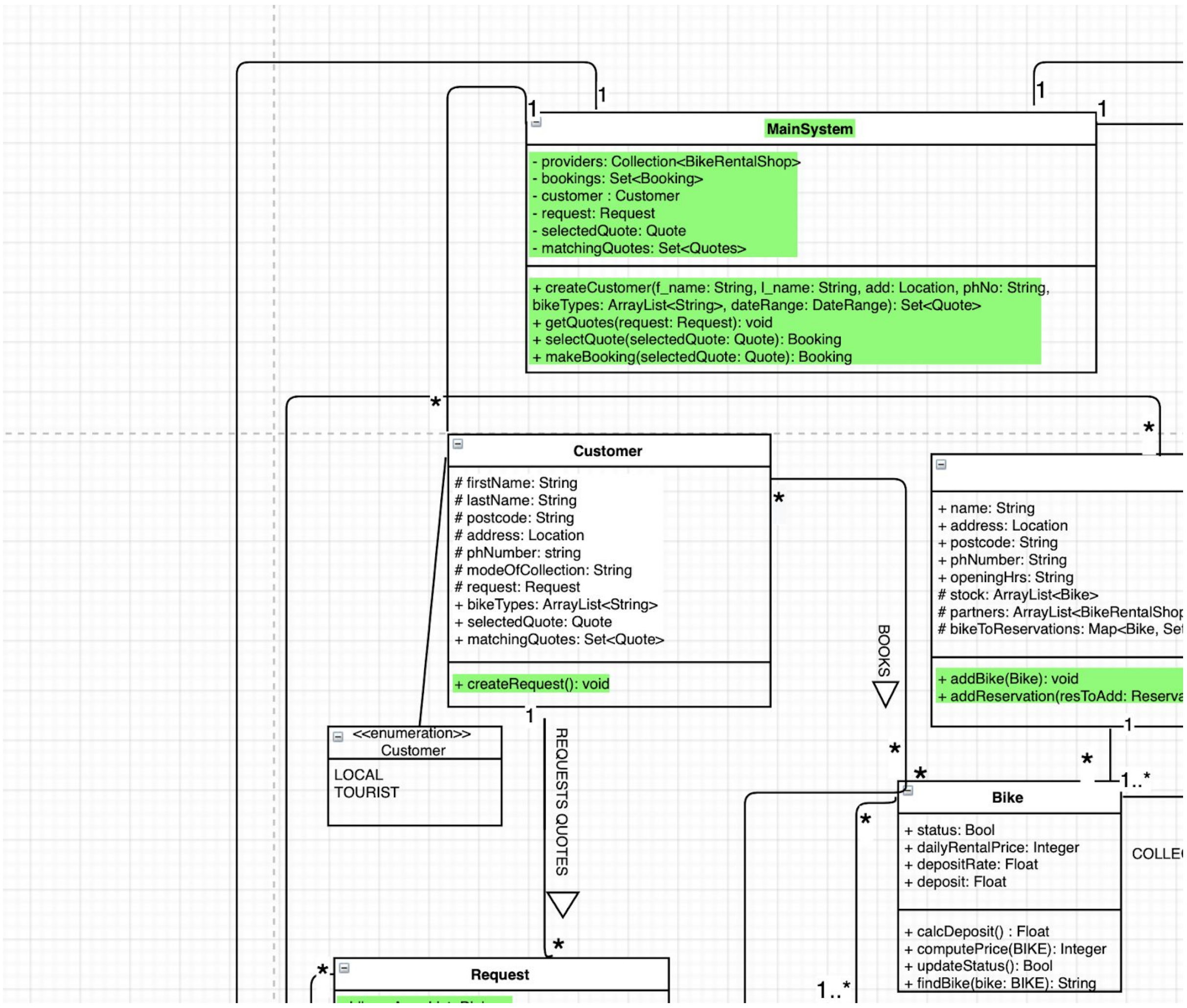
Coursework 2

- ❑ Daniel Wilks (UUN: s1851664)
- ❑ Eshaan Manglik (UUN: s1892592)

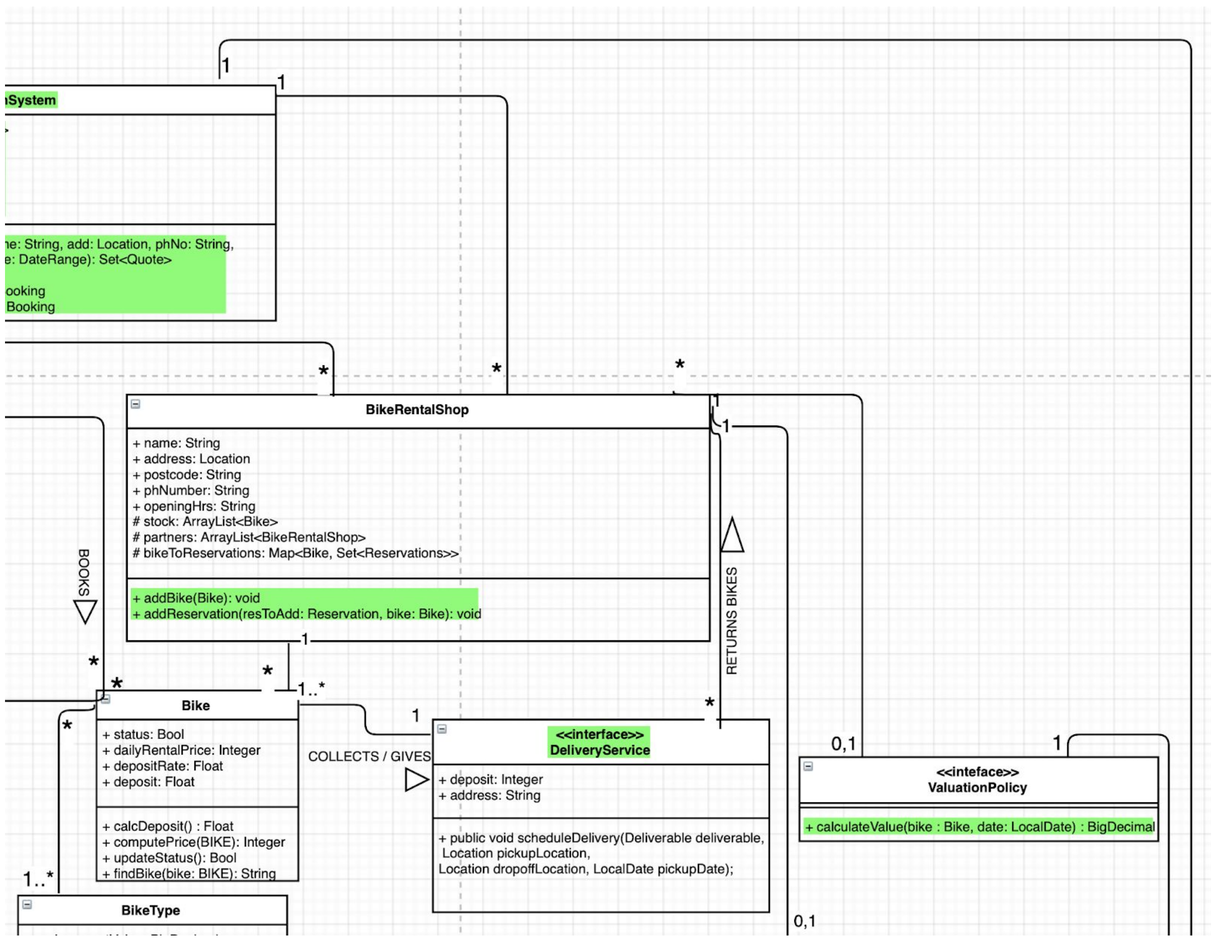
Q 2.2.1 UML class model


MainSystem
- providers: Collection<BikeRentalShop>
- bookings: Set<Booking>
- customer : Customer
- request: Request
- selectedQuote: Quote
- matchingQuotes: Set<Quotes>
+ createCustomer(f_name: String, l_name: String, add: Location, phNo: String, bikeTypes: ArrayList<String>, dateRange: DateRange): Set<Quote>
+ getQuotes(request: Request): void
+ selectQuote(selectedQuote: Quote): Booking
+ makeBooking(selectedQuote: Quote): Booking
Customer
# firstName: String
# lastName: String
# postcode: String
# address: Location
# phNumber: string
# modeOfCollection: String
# request: Request
+ bikeTypes: ArrayList<String>
+ selectedQuote: Quote
+ matchingQuotes: Set<Quote>
+ createRequest(): void
<<enumeration>>
Customer
LOCAL
TOURIST
REQUESTS QUOTES
BOOKS
BikeRentalShop
+ name: String
+ address: Location
+ postcode: String
+ phNumber: String
+ openingHrs: String
# stock: ArrayList<Bike>
# partners: ArrayList<BikeRentalShop>
# bikeToReservations: Map<Bike, Set<Reservations>>
+ addBike(Bike): void
+ addReservation(resToAdd: Reservation, bike: Bike): void
RETURNS BIKES
Bike
+ status: Bool
+ dailyRentalPrice: Integer
+ depositRate: Float
+ deposit: Float
+ calcDeposit() : Float
+ computePrice(BIKE): Integer
+ updateStatus(): Bool
+ findBike(bike: BIKE): String
COLLECTS / GIVES
<<interface>>
DeliveryService
+ deposit: Integer
+ address: String
+ public void scheduleDelivery(Deliverable deliverable, Location pickupLocation, Location dropoffLocation, LocalDate pickupDate);
<<inteface>>
ValuationPolicy
+ calculateValue(bike : Bike, date: LocalDate) : BigDecimal
Request
- bikes: ArrayList<Biek>
- dateRange: DateRange
- location: Location
- modeOfCollection: String
BikeType
+ replacementValue: BigDecimal
+ getReplacementValue(): BigDecimal
+ setReplacementValue(BigDecimal): void
<<interface>>
PricingPolicy
+ setDailyRentalPrice(bikeType : BikeType, price : BigDecimal)
+ calculatePrice(bikes : Collection<Bike>, date : DateRange): BigDecimal
QuoteController
- providers: ArrayList<BikeRentalShop>
- macthingQuotes: Set<Quote>
+ findMatchingQuote(request: Request): Set<Quote>
Booking
- orderNumber: Integer
- orderSummary: String
- deposit: Double
- totalPrice: Double
- delivery: String
+ shop: BikeRentalShop
+ dateRange: Integer
+ pickUpLoc: Location
+ bikes: ArrayList<Bike>
Bank
- cardNo: String
- cardName: String
- cvv: String
- valid: Bool
+ validCardDetails(): Bool
- makeTransaction(valid): void
Quote
- shop: BikeRentalShop
- dateRange: DateRange
- totalPrice: Double
- deposit: Double
- bikes: ArrayList<Bike>
Reservation
# customer: Customer
# dateRange: DateRange
1
*
1..*
0,1

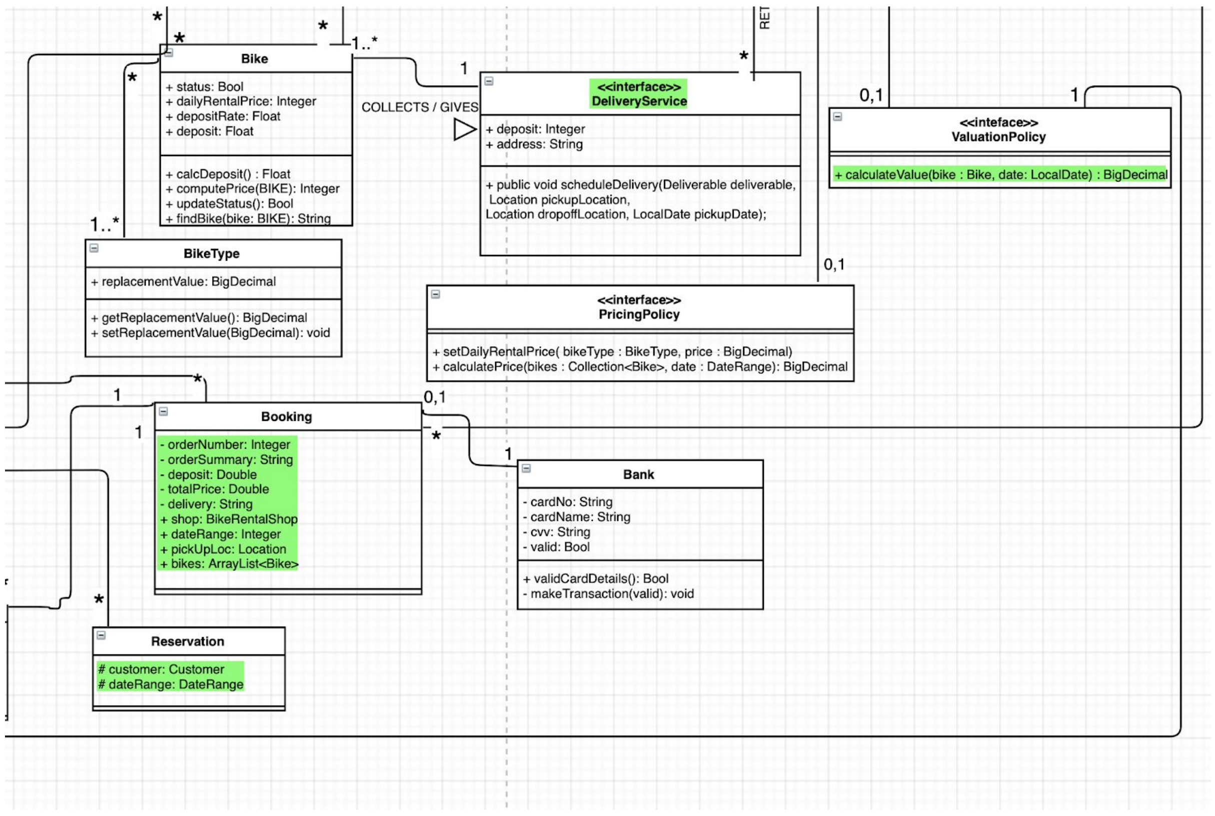
Incase the image is not readable, we have provided zoom in images of each component below.


MainSystem
- providers: Collection<BikeRentalShop>
- bookings: Set<Booking>
- customer : Customer
- request: Request
- selectedQuote: Quote
- matchingQuotes: Set<Quotes>
+ createCustomer(f_name: String, l_name: String, add: Location, phNo: String, bikeTypes: ArrayList<String>, dateRange: DateRange): Set<Quote>
+ getQuotes(request: Request): void
+ selectQuote(selectedQuote: Quote): Booking
+ makeBooking(selectedQuote: Quote): Booking
Customer
# firstName: String
# lastName: String
# postcode: String
# address: Location
# phNumber: string
# modeOfCollection: String
# request: Request
+ bikeTypes: ArrayList<String>
+ selectedQuote: Quote
+ matchingQuotes: Set<Quote>
+ createRequest(): void
<<enumeration>>
Customer
LOCAL
TOURIST
+ name: String
+ address: Location
+ postcode: String
+ phNumber: String
+ openingHrs: String
# stock: ArrayList<Bike>
# partners: ArrayList<BikeRentalShop
# bikeToReservations: Map<Bike, Set
+ addBike(Bike): void
+ addReservation(resToAdd: Reserva
Bike
+ status: Bool
+ dailyRentalPrice: Integer
+ depositRate: Float
+ deposit: Float
+ calcDeposit() : Float
+ computePrice(BIKE): Integer
+ updateStatus(): Bool
+ findBike(bike: BIKE): String
Request
REQUESTS QUOTES
BOOKS
COLLE
1
1
1
1
*
*
*
1
*
*
*
1
*
1..*
*
1..*
*

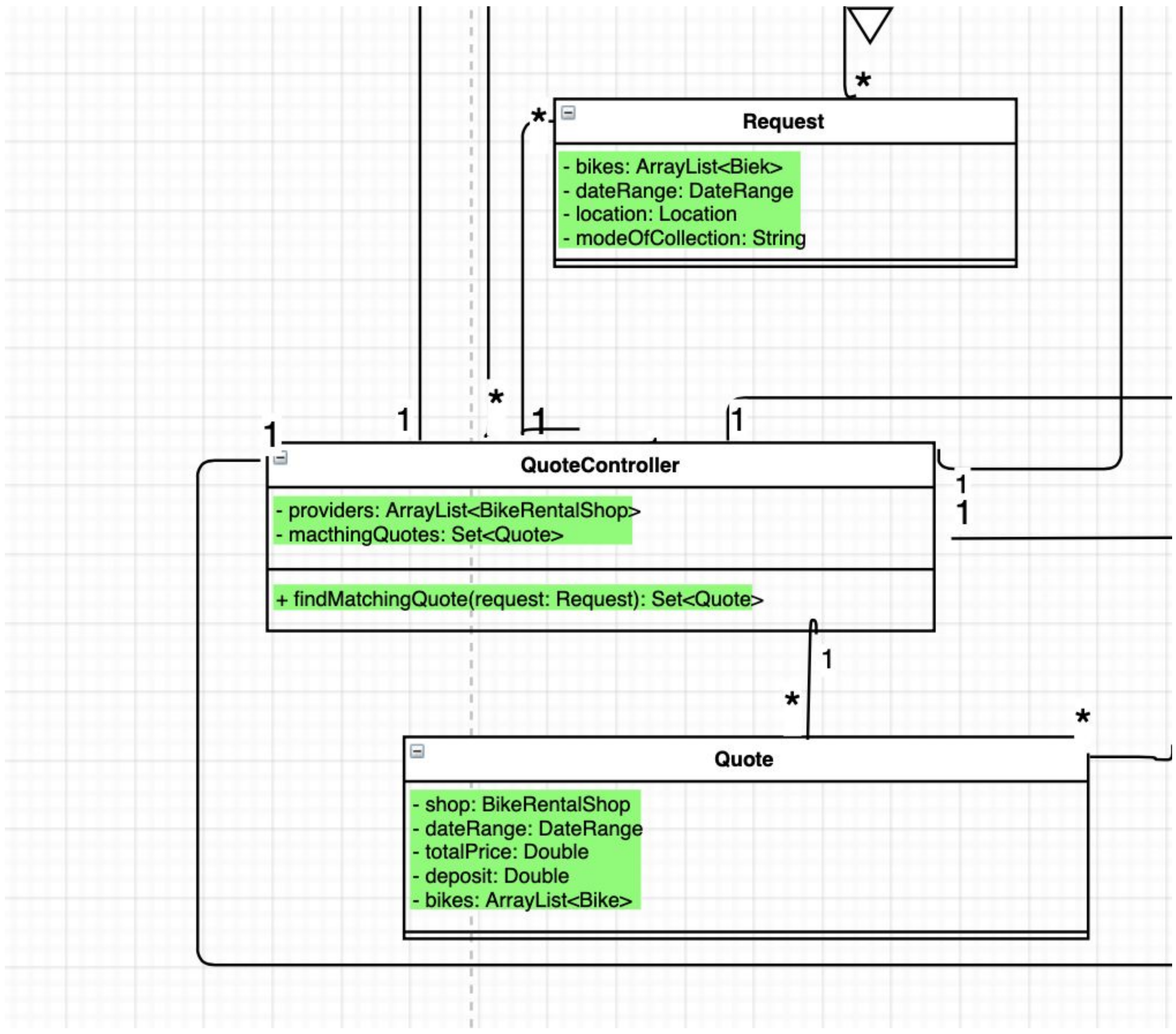
Top Left


1
1
System
e: String, add: Location, phNo: String,
e: DateRange): Set<Quote>
ooking
Booking
*
*
*
BikeRentalShop
+ name: String
+ address: Location
+ postcode: String
+ phNumber: String
+ openingHrs: String
# stock: ArrayList<Bike>
# partners: ArrayList<BikeRentalShop>
# bikeToReservations: Map<Bike, Set<Reservations>>
+ addBike(Bike): void
+ addReservation(resToAdd: Reservation, bike: Bike): void
1
1
BOOKS
RETURNS BIKES
1
*
*
*
1..*
*
Bike
+ status: Bool
+ dailyRentalPrice: Integer
+ depositRate: Float
+ deposit: Float
+ calcDeposit() : Float
+ computePrice(BIKE): Integer
+ updateStatus(): Bool
+ findBike(bike: BIKE): String
1
COLLECTS / GIVES
*
<<interface>>
DeliveryService
+ deposit: Integer
+ address: String
+ public void scheduleDelivery(Deliverable deliverable,
Location pickupLocation,
Location dropoffLocation, LocalDate pickupDate);
0,1
1
<<inteface>>
ValuationPolicy
+ calculateValue(bike : Bike, date: LocalDate) : BigDecimal
1..*
BikeType
0,1

Top Right



Bottom Right



Bottom Left

Q 2.2.2 High-level description

The system in use starts with the customer inputting personal details. For the customer, we decided to use an enumeration class because it allows us to have only one class instead of two (Local and Tourist). This increases readability.

We created a class for the bike rental shops with their details as instance variables. We created a variable 'stock' which is a collection of the bikes that the provider has. To create the relation between bike shop partners we created a variable 'partners' that is a collection of all the partners (which are the partners for that particular Bike Rental Shop).

The customer then requests quotes. We created a function getQuotes() inside the MainSystem class where the instance of the customer is created and a request class is created which contains the customers rental needs. The getQuotes() returns a collection of Quote objects from the QuoteController class which is stored in an instance variable called ‘matchingQuotes’. We did this as this allows us to easily access quotes that suit the customer. There can only be one Bike provider per quote.

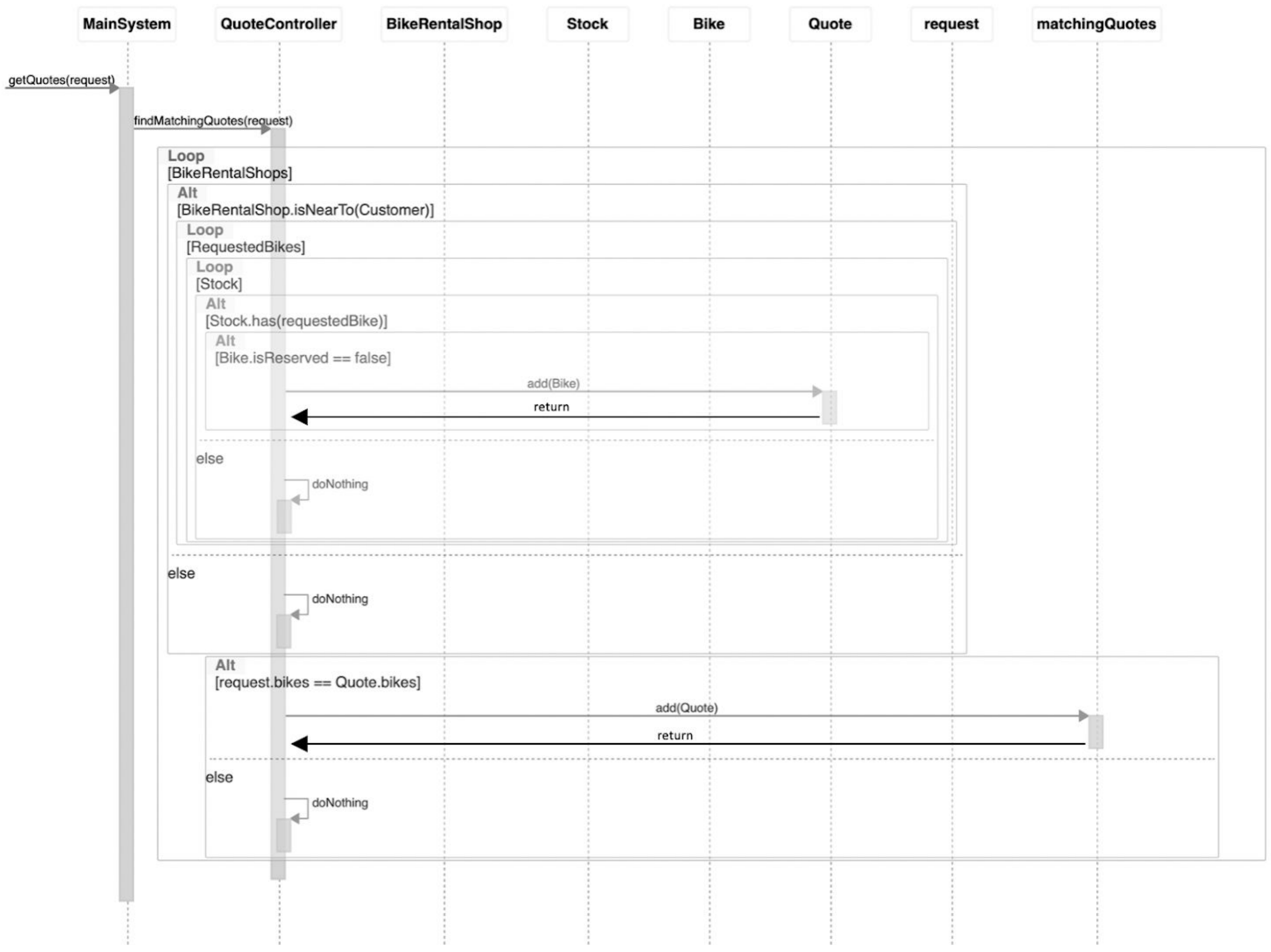
After the request, the quote controller is invoked which first checks if the bike provider is near the location of the customer and then checks if each bike provider has the desired bikes available without any overlaps and if the bikes are available the quote class is invoked in which the quote gets created and added to a collection of quotes called ‘matchingQuotes’ and then further returned to the main system.

Ones the customer selects a quote and a booking class is invoked by the MainSystem class. They make an online payment by entering their card details which are checked to see if they are valid. The bank class is where the actual transaction is processed and the summary of the booking is displayed. The delivery is scheduled if the customer has requested for a delivery. Which further invokes the reservation class only if the payment has been made and maps it to that particular bike provider. This is where the bikes get reserved for the customer and the status of them are updated to RESERVED. We used an enum class for the BikeStatus and BookingStatus as it makes the status more clear and increases readability. In the reservation class we have an instance of class Customer which allows us to access the details of the customer from the reservation class.

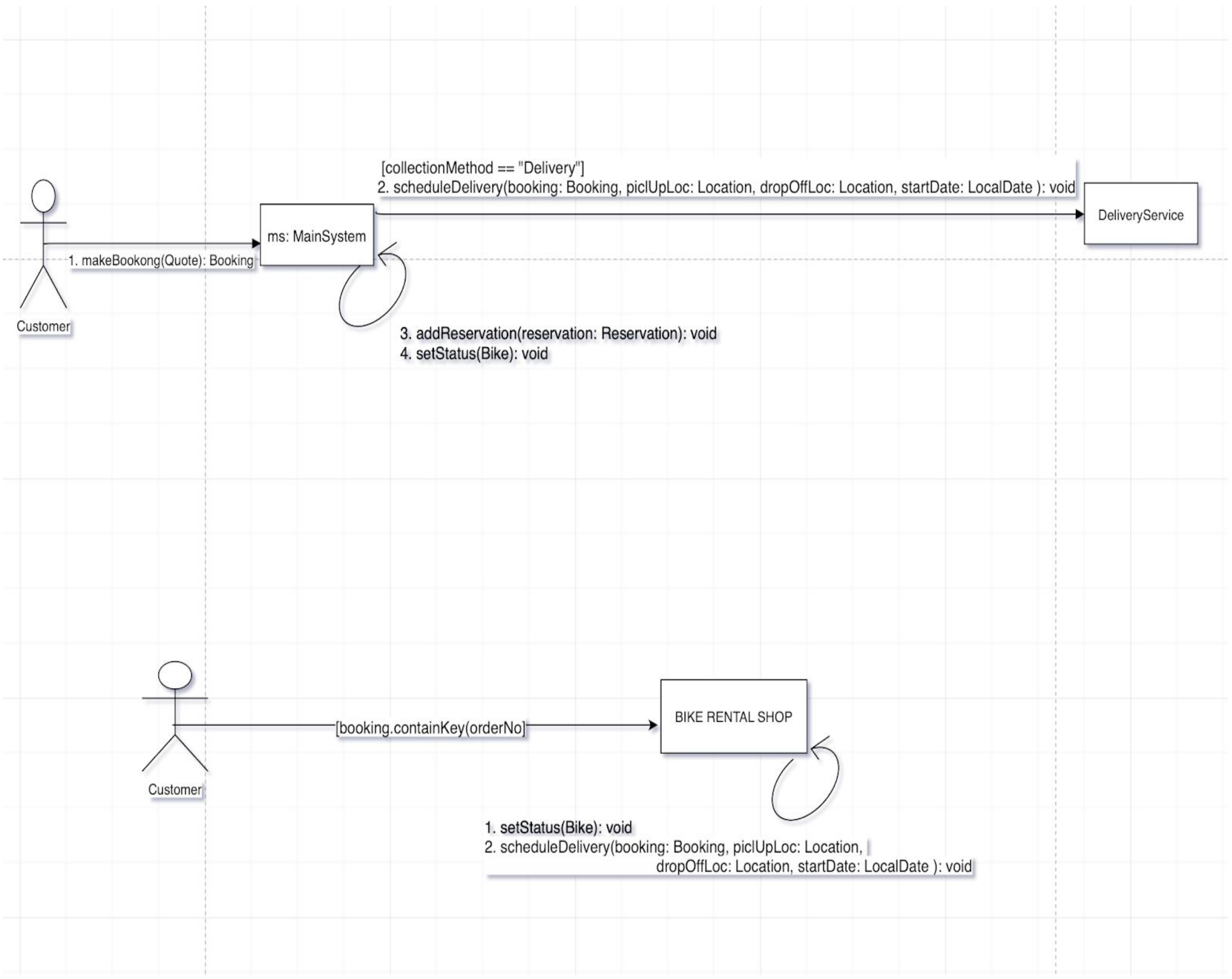
We also created a class for the delivery drivers which have the ability to collect the bikes and update the status of said bikes.

In the returning of the bikes the status is changed to AVAILABLE if they are returned to the Original provider and if they are returned to the partner provider the status is changed to PARTNER_TO_ORIGPROVIDER and a delivery is scheduled using the deliveryService class.

Q 2.3.1 UML sequence diagram (The diagram is completely new)



Q 2.3.2 UML communication diagram


[collectionMethod == "Delivery"]
2. scheduleDelivery(booking: Booking, piclUpLoc: Location, dropOffLoc: Location, startDate: LocalDate): void
DeliveryService
ms: MainSystem
1. makeBookong(Quote): Booking
Customer
3. addReservation(reservation: Reservation): void
4. setStatus(Bike): void
[booking.containKey(orderNo]
BIKE RENTAL SHOP
Customer
1. setStatus(Bike): void
2. scheduleDelivery(booking: Booking, piclUpLoc: Location,
dropOffLoc: Location, startDate: LocalDate): void

Q 2.4 Conformance to requirements

We removed the use case ‘Make Payment’. When making the use case diagram we found that the functionality we wanted could be captured in a method (instead of the class ‘Make Payment’) inside the ‘Booking’ class which is executed after the details of the customer’s card are validated by the bank.

Q 2.5.3 Design extensions

• Specify interfaces for pricing policies and deposit/valuation policies

We have added an interface called ‘Pricing Policies’ in which we have methods like computeDeposit() and other methods which the contractor can add pricing policies and calculate deposit and makes make it possible to implement the newly planned pricing and deposit policies of existing bike providers.

• Integrate interfaces into class diagram

We integrated the interface into our class diagram. We believe we adequately created functionality for bike rental shops to have custom pricing policies.

Q 2.6 Self-assessment

Q 2.2.1 Static model 25%

• Make correct use of UML class diagram notation 5%

5%, As we used the correct notation for the UML diagram

• Split the system design into appropriate classes 5%

4%, We split up the design into all the appropriate necessary classes like customer, bike rental shop, etc but however we might have used the most optimal classes to reduce coupling.

• Include necessary attributes and methods for use cases 5%

3%, We included all the attributes and methods we considered necessary however we could reduce coupling in certain methods.

• Represent associations between classes 5%

5%, We notated all the associations between the classes we created.

• Follow good software engineering practices 5%

4%, We used appropriate data types like maps and collections to represent good software practices that makes the code more accessible.

Q 2.2.2 High-level description 15

• Describe/clarify key components of design 10%

8%, We described the key classes, attributes and methods of our design in detail. We might not have provided enough detail on a few components.

• Discuss design choices/resolution of ambiguities 5%

4%, We described and explained our design choices to an adequate level. However, we might not have resolved all ambiguities in our design.

Q 2.3.1 UML sequence diagram 20%

- Correctly use of UML sequence diagram notation 5%

5%, We used the correct UML sequence diagram notation.

- **Cover class interactions involved in UML sequence diagram 10%**

8%, We almost covered all the interaction between the classes including the return interactions.

- **Represent optional, alternative, and iterative behaviour where appropriate 5%**

3%, We adequately showed where iterative and the alternative behaviour by making use of if and the loop frames, but we could have missed the optional behaviour.

Q 2.3.2 UML communication diagram 15%

- **Communication diagram for *return bikes to original provider* use case 8%**

7%, We covered almost all the cases like when the delivery driver returns the bikes or when the customer himself does it.

- **Communication diagram for *book quote* use case 7%**

6%, We adequately showed the interactions between the involved classes.

Q 2.4 Conformance to requirements 5%

• **Ensure conformance to requirements and discuss issues 5%**

3%, We have made changes to our use cases and use case diagram by removing the 'Make Payment' use case but we might have missed some other changes that could make the use case diagram and the class model more consistent with the system requirements.

Q 2.5.3 Design extensions 10%

- **Specify interfaces for pricing policies and deposit/valuation policies 3%**
- **Integrate interfaces into class diagram 7%**

We integrated the interface into our class diagram. We believe we adequately created functionality for bike rental shops to have custom pricing policies.

Q 2.6 Self-assessment 10%

- **Justification of good software engineering practice 5%**

We used a simple design so that the maintainability and readability of our design is high. We also made the names of variables/ methods self explanatory. Instead of using multiple variables for same entity we used interfaces such as maps and collections. We have tried to keep coupling low in our classes by creating functions that have little dependence on other functions. We included an enumeration class as to make our design more readable. We made use of interfaces so that in the future our code could be updated more easily.