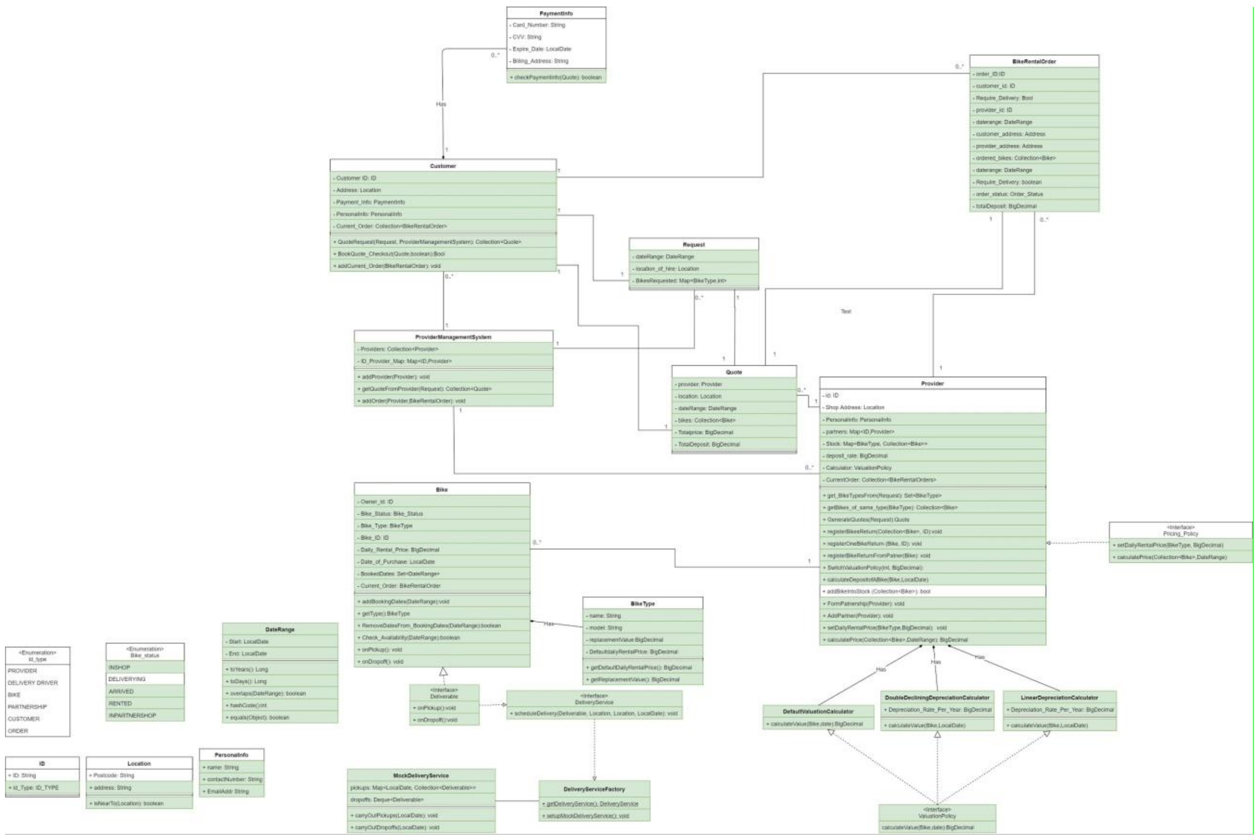
SOFTWARE ENGINEERING

CW2 Design

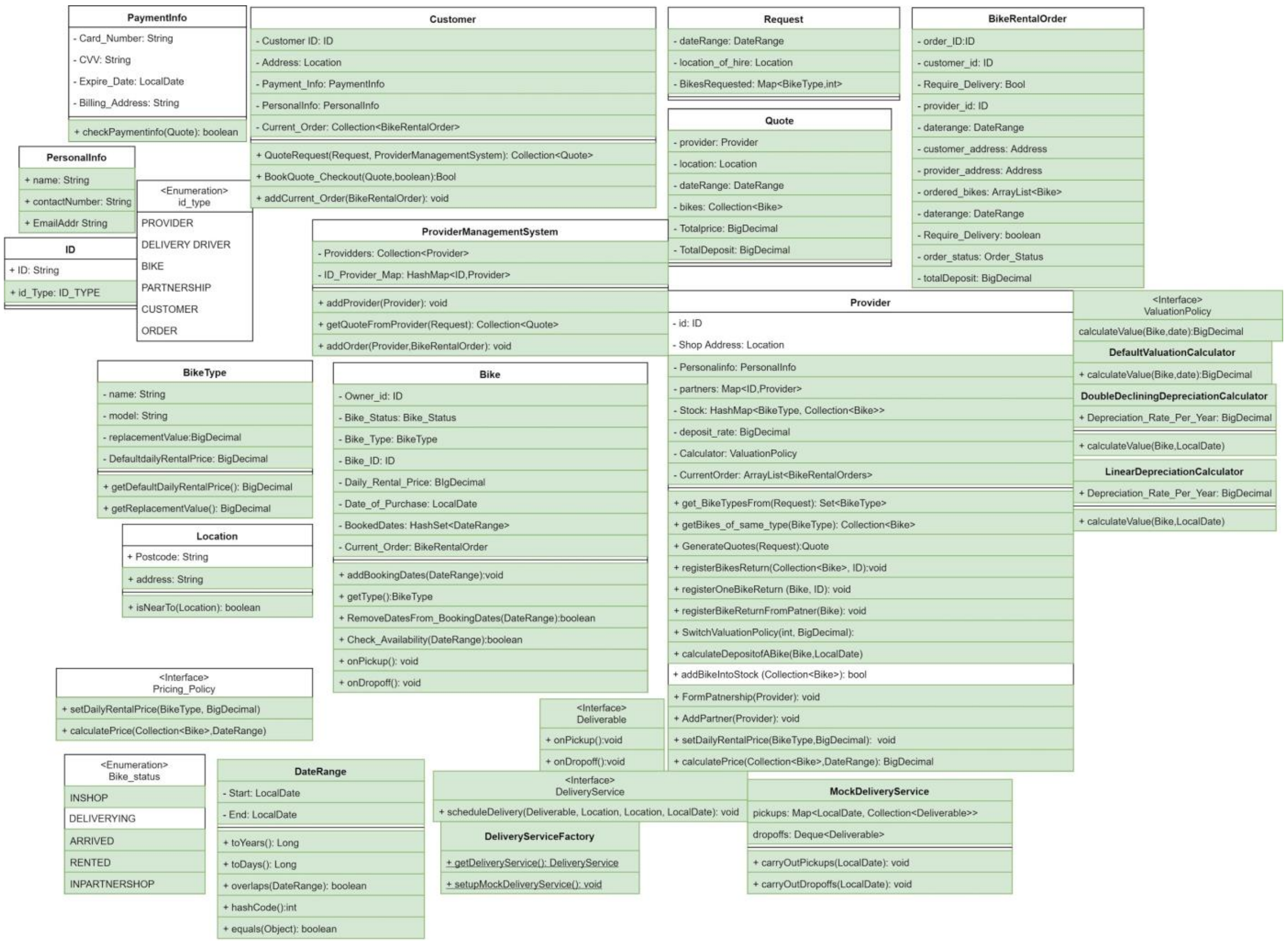
s1857043
s1864509

TABLE OF CONTENT

Class Diagram


PaymentInfo
- Card_Number: String
- CVV: String
- Expire_Date: LocalDate
- Billing_Address: String
+ checkPaymentInfo(Quote): boolean
Has
0..*
1
Customer
- Customer ID: ID
- Address: Location
- Payment_Info: PaymentInfo
- PersonalInfo: PersonalInfo
- Current_Order: Collection<BikeRentalOrder>
+ QuoteRequest(Request, ProviderManagementSystem): Collection<Quote>
+ BookQuote_Checkout(Quote,boolean):Bool
+ addCurrent_Order(BikeRentalOrder): void
BikeRentalOrder
- order_ID:ID
- customer_id: ID
- Require_Delivery: Bool
- provider_id: ID
- daterange: DateRange
- customer_address: Address
- provider_address: Address
- ordered_bikes: Collection<Bike>
- daterange: DateRange
- Require_Delivery: boolean
- order_status: Order_Status
- totalDeposit: BigDecimal
Request
- dateRange: DateRange
- location_of_hire: Location
- BikesRequested: Map<BikeType,int>
Text
ProviderManagementSystem
- Providers: Collection<Provider>
- ID_Provider_Map: Map<ID,Provider>
+ addProvider(Provider): void
+ getQuoteFromProvider(Request): Collection<Quote>
+ addOrder(Provider,BikeRentalOrder): void
Quote
- provider: Provider
- location: Location
- dateRange: DateRange
- bikes: Collection<Bike>
- Totalprice: BigDecimal
- TotalDeposit: BigDecimal
Provider
- id: ID
- Shop Address: Location
- PersonalInfo: PersonalInfo
- partners: Map<ID,Provider>
- Stock: Map<BikeType, Collection<Bike>>
- deposit_rate: BigDecimal
- Calculator: ValuationPolicy
- CurrentOrder: Collection<BikeRentalOrders>
+ get_BikeTypesFrom(Request): Set<BikeType>
+ getBikes_of_same_type(BikeType): Collection<Bike>
+ GenerateQuotes(Request):Quote
+ registerBikesReturn(Collection<Bike>, ID):void
+ registerOneBikeReturn (Bike, ID): void
+ registerBikeReturnFromPatner(Bike): void
+ SwitchValuationPolicy(int, BigDecimal):
+ calculateDepositofABike(Bike,LocalDate)
+ addBikeIntoStock (Collection<Bike>): bool
+ FormPatnership(Provider): void
+ AddPartner(Provider): void
+ setDailyRentalPrice(BikeType,BigDecimal): void
+ calculatePrice(Collection<Bike>,DateRange): BigDecimal
<Interface>
Pricing_Policy
+ setDailyRentalPrice(BikeType, BigDecimal)
+ calculatePrice(Collection<Bike>,DateRange)
Bike
- Owner_Id: ID
- Bike_Status: Bike_Status
- Bike_Type: BikeType
- Bike_ID: ID
- Daily_Rental_Price: BigDecimal
- Date_of_Purchase: LocalDate
- BookedDates: Set<DateRange>
- Current_Order: BikeRentalOrder
+ addBookingDates(DateRange):void
+ getType():BikeType
+ RemoveDatesFrom_BookingDates(DateRange):boolean
+ Check_Availability(DateRange):boolean
+ onPickup(): void
+ onDropoff(): void
BikeType
- name: String
- model: String
- replacementValue:BigDecimal
- DefaultdailyRentalPrice: BigDecimal
+ getDefaultDailyRentalPrice(): BigDecimal
+ getReplacementValue(): BigDecimal
<Interface>
Deliverable
+ onPickup():void
+ onDropoff():void
<Interface>
DeliveryService
+ scheduleDelivery(Deliverable, Location, Location, LocalDate): void
MockDeliveryService
pickups: Map<LocalDate, Collection<Deliverable>>
dropoffs: Deque<Deliverable>
+ carryOutPickups(LocalDate): void
+ carryOutDropoffs(LocalDate): void
DeliveryServiceFactory
+ getDeliveryService(): DeliveryService
+ setupMockDeliveryService(): void
DefaultValuationCalculator
+ calculateValue(Bike,date):BigDecimal
DoubleDecliningDepreciationCalculator
+ Depreciation_Rate_Per_Year: BigDecimal
+ calculateValue(Bike,LocalDate)
LinearDepreciationCalculator
+ Depreciation_Rate_Per_Year: BigDecimal
+ calculateValue(Bike,LocalDate)
<Interface>
ValuationPolicy
calculateValue(Bike,date):BigDecimal
<Enumeration>
Id_type
PROVIDER
DELIVERY DRIVER
BIKE
PARTNERSHIP
CUSTOMER
ORDER
<Enumeration>
Bike_status
INSHOP
DELIVERYING
ARRIVED
RENTED
INPARTNERSHOP
DateRange
- Start: LocalDate
- End: LocalDate
+ toYears(): Long
+ toDays(): Long
+ overlaps(DateRange): boolean
+ hashCode():int
+ equals(Object): boolean
ID
+ ID: String
+ Id_Type: ID_TYPE
Location
+ Postcode: String
+ address: String
+ isNearTo(Location): boolean
PersonalInfo
+ name: String
+ contactNumber: String
+ EmailAddr: String

Class Diagram(Simplified)


PaymentInfo
- Card_Number: String
- CVV: String
- Expire_Date: LocalDate
- Billing_Address: String
+ checkPaymentinfo(Quote): boolean
Customer
- Customer ID: ID
- Address: Location
- Payment_Info: PaymentInfo
- PersonalInfo: PersonalInfo
- Current_Order: Collection<BikeRentalOrder>
+ QuoteRequest(Request, ProviderManagementSystem): Collection<Quote>
+ BookQuote_Checkout(Quote,boolean):Bool
+ addCurrent_Order(BikeRentalOrder): void
Request
- dateRange: DateRange
- location_of_hire: Location
- BikesRequested: Map<BikeType,int>
BikeRentalOrder
- order_ID:ID
- customer_id: ID
- Require_Delivery: Bool
- provider_id: ID
- daterange: DateRange
- customer_address: Address
- provider_address: Address
- ordered_bikes: ArrayList<Bike>
- daterange: DateRange
- Require_Delivery: boolean
- order_status: Order_Status
- totalDeposit: BigDecimal
Quote
- provider: Provider
- location: Location
- dateRange: DateRange
- bikes: Collection<Bike>
- Totalprice: BigDecimal
- TotalDeposit: BigDecimal
PersonalInfo
+ name: String
+ contactNumber: String
+ EmailAddr String
<Enumeration>
id_type
PROVIDER
DELIVERY DRIVER
BIKE
PARTNERSHIP
CUSTOMER
ORDER
ID
+ ID: String
+ id_Type: ID_TYPE
ProviderManagementSystem
- Providders: Collection<Provider>
- ID_Provider_Map: HashMap<ID,Provider>
+ addProvider(Provider): void
+ getQuoteFromProvider(Request): Collection<Quote>
+ addOrder(Provider,BikeRentalOrder): void
Provider
- id: ID
- Shop Address: Location
- PersonalInfo: PersonalInfo
- partners: Map<ID,Provider>
- Stock: HashMap<BikeType, Collection<Bike>>
- deposit_rate: BigDecimal
- Calculator: ValuationPolicy
- CurrentOrder: ArrayList<BikeRentalOrders>
+ get_BikeTypesFrom(Request): Set<BikeType>
+ getBikes_of_same_type(BikeType): Collection<Bike>
+ GenerateQuotes(Request):Quote
+ registerBikesReturn(Collection<Bike>, ID):void
+ registerOneBikeReturn (Bike, ID): void
+ registerBikeReturnFromPatner(Bike): void
+ SwitchValuationPolicy(int, BigDecimal):
+ calculateDepositofABike(Bike,LocalDate)
+ addBikeIntoStock (Collection<Bike>): bool
+ FormPatnership(Provider): void
+ AddPartner(Provider): void
+ setDailyRentalPrice(BikeType,BigDecimal): void
+ calculatePrice(Collection<Bike>,DateRange): BigDecimal
<Interface>
ValuationPolicy
calculateValue(Bike,date):BigDecimal
DefaultValuationCalculator
+ calculateValue(Bike,date):BigDecimal
DoubleDecliningDepreciationCalculator
+ Depreciation_Rate_Per_Year: BigDecimal
+ calculateValue(Bike,LocalDate)
LinearDepreciationCalculator
+ Depreciation_Rate_Per_Year: BigDecimal
+ calculateValue(Bike,LocalDate)
BikeType
- name: String
- model: String
- replacementValue:BigDecimal
- DefaultdailyRentalPrice: BigDecimal
+ getDefaultDailyRentalPrice(): BigDecimal
+ getReplacementValue(): BigDecimal
Bike
- Owner_id: ID
- Bike_Status: Bike_Status
- Bike_Type: BikeType
- Bike_ID: ID
- Daily_Rental_Price: BigDecimal
- Date_of_Purchase: LocalDate
- BookedDates: HashSet<DateRange>
- Current_Order: BikeRentalOrder
+ addBookingDates(DateRange):void
+ getType():BikeType
+ RemoveDatesFrom_BookingDates(DateRange):boolean
+ Check_Availability(DateRange):boolean
+ onPickup(): void
+ onDropoff(): void
Location
+ Postcode: String
+ address: String
+ isNearTo(Location): boolean
<Interface>
Pricing_Policy
+ setDailyRentalPrice(BikeType, BigDecimal)
+ calculatePrice(Collection<Bike>,DateRange)
<Interface>
Deliverable
+ onPickup():void
+ onDropoff():void
<Enumeration>
Bike_status
INSHOP
DELIVERYING
ARRIVED
RENTED
INPARTNERSHOP
DateRange
- Start: LocalDate
- End: LocalDate
+ toYears(): Long
+ toDays(): Long
+ overlaps(DateRange): boolean
+ hashCode():int
+ equals(Object): boolean
<Interface>
DeliveryService
+ scheduleDelivery(Deliverable, Location, Location, LocalDate): void
DeliveryServiceFactory
+ getDeliveryService(): DeliveryService
+ setupMockDeliveryService(): void
MockDeliveryService
pickups: Map<LocalDate, Collection<Deliverable>>
dropoffs: Deque<Deliverable>
+ carryOutPickups(LocalDate): void
+ carryOutDropoffs(LocalDate): void

Changes we have made:

1. Replace name, Phone No. and Email address with PersonalInfo, consisting with the design and code.
2. Add attribute “Partners”, helping Provider to quickly find the correct partner(owner) to whom the bike needed to return in the use case of “Return bike to partner” .
3. Add ‘ValutionPolicyCalculator’, helping bike providers to calculate deposit of bikes in a more flexible way. For example, they can choose which way to estimate the replacement value of a bike: Linear, double declining or default.
4. Add Current Orders for provider to use API from BIKERENTALORDER to modify order status in use case of recording bike return.
5. Add Date_of_Purchase to help provider to estimate the replacement value of bikes based on its age.
6. Add BookedDates to show a bike’s availability in a long period of time.
7. Add Customer_id,provider_id, Customer_Address, Provider_Address, Ordered_Bikes and DateRange to confirm details of the whole rental order and provide useful information to both provider and customer.
8. Add TotalDeposit to indicate how much deposit that customer have to pay before getting the bike(s) or how much provider /partner has to return when the customer returns the bike(s);
9. Reduced to 4 different status: [DELIVERING], [ARRIVED], [RENTED], [INPARTNERSHOP] to consist with UML class diagram
10. Reduced to status of [BOOKED], [BIKEDELIVERYING],[BIKEARRIVED],[BIKERETURNEDTOPARTNERSHOP], [COMPLETED] to consist with UML class diagram

High-level Description

Big picture:

‘Bike’, ‘Provider’, ‘Customer’, ‘BikeRentalOrder’, ‘Request’, ‘Quote’, ‘ProviderManagementSystem’ are the 7 core classes in the UML class diagram, sending calls to one another in a relatively high frequency. On the other hand, we also have other classes, such as **‘payment_info’, ‘DateRange’** and ‘**Location’**, which play essential roles of storing encapsulated data. Altering their attributes will not affect the rest of the system. They have more independence as compared to the seven core classes.

Overview of 7 core classes:

*Some attributes and methods may not be covered in the following description as they can be clearly explained by their names, e.g. Daily_Rental_Price: Double.

*Attributes in blue

*Methods in brown

Class name	Description
Bike	‘Bike’ class has an attribute ‘BookedDates’ (Set<DateRange>) storing all the DateRanges it was booked in. Then, when Bike is called by GetQuote use case to ‘Check_Availability(DateRange)’, it will take the DateRange in the function and check if the DateRange overlaps with any in ‘BookedDates’ to ensure its availability.

Customer	The ‘Customer’ class contains 5 attributes storing information about customer’s ‘PersonalInfo’, ‘ID’ that was assigned when the class is created. ‘Address’ for potential delivery service, ‘Payment_Info’ and ‘Current_Order’ (Collection<BikeRentalOrder>). It provides users with 3 methods’QuoteRequest(Request, ProviderManagementSystem)’ to search among providers for satisfactory quotes, ‘BookQuote_CheckOut’ to book the desired quote and ‘addCurrent_Order(BikeRentalOrder)’ to update ‘Current_Order’ attribute when ‘BookQuote_CheckOut’ initializes.
Provider	The ‘Provider’ class contains attribute ‘partners’ (Map<ID,Provider>). This is because every provider has to possess complete information of their partners in case they need to return the bike to them. The ‘Calculator’ (ValuationPolicy) attribute contains ValuationPolicy of how the deposit should be calculated. It contains an method named ‘GenerateQuote(Request)’. This API takes information of ‘BikeType’ and number of ‘Bike’ requested and collect all bikes that satisfies the request, forming a quote and return it.

BikeRentalOrder	The ‘BikeRentalOrder’ class gathers information from ‘Quote’ and ’Customer’ stores these information within it. It has an attribute named ‘Require_Delivery’ stored in boolean. It was assigned as this class is initialized. If it is true, the class will automatically call DeliveryServiceFactory to generate a DeliveryService.
Request	‘Request’ class encapsulates information when customers key in their requirements, this will be the constraints to help system to find the most appropriate quotes for customers.
Quote	This class contains information about Bikes and providers that satisfied customer’s request. Also, it contains Price and deposit that can be displayed to customers. In GetQuote use case, these quotes are created and returned for customers to choose from.

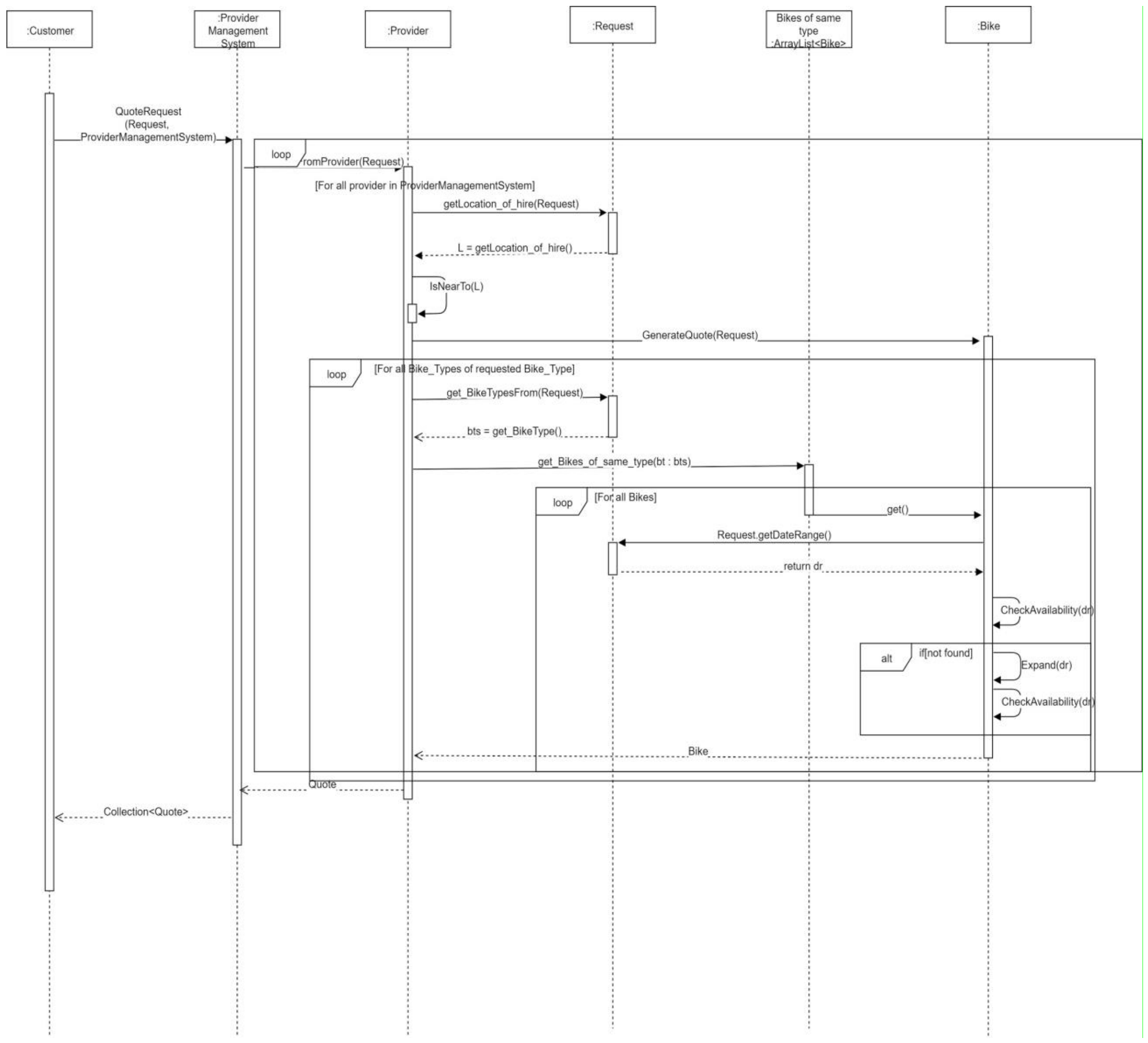
ProviderManagementSystem	The 'ProviderManagementSystem' class has 2 attributes: 'Providers' (Collection<Provider>) and 'ID_Provider_Map'(HashMap<ID, Provider>). Storing information about every provider allowing Get_quote to be more straightforward.

Discussion of choice:

Choice	Reason
Make BikeType a class	A BikeType class could store multiple data such as name/model, replacement value and default DailyPrice. It can also provide functions for Bike or other classes to access its encapsulated data.

Use Map<BikeType,Collection<Bike>> Stock to store bike objects in Provider class	One advantage of using map is that it classifies each bike by BikeType. This helps system to find desired bikes from a Partner class in a more efficient way. For instance, customer will input BikeType to the system in order to find relevant quotes and the system could easily get a collection of bikes from a provider, by using the API provided by Map, Stock.get(BikeType).
Use BigDecimal to represent price, deposit rate and depreciation rate.	Comparing to double, It is more accurate and safer to use.
Make DateRange a class	DateRange represents a period of time. It is very useful in the scenario of getting quotes. For example, we can use one DateRange object to represent the period of time that customer wants to rent a bike and a collection of DateRange objects to express the periods when the bike is not available. We can simply use overlap(DateRange) to find if the bike is available in the requested dates.

Sequence Diagram： Get Quote

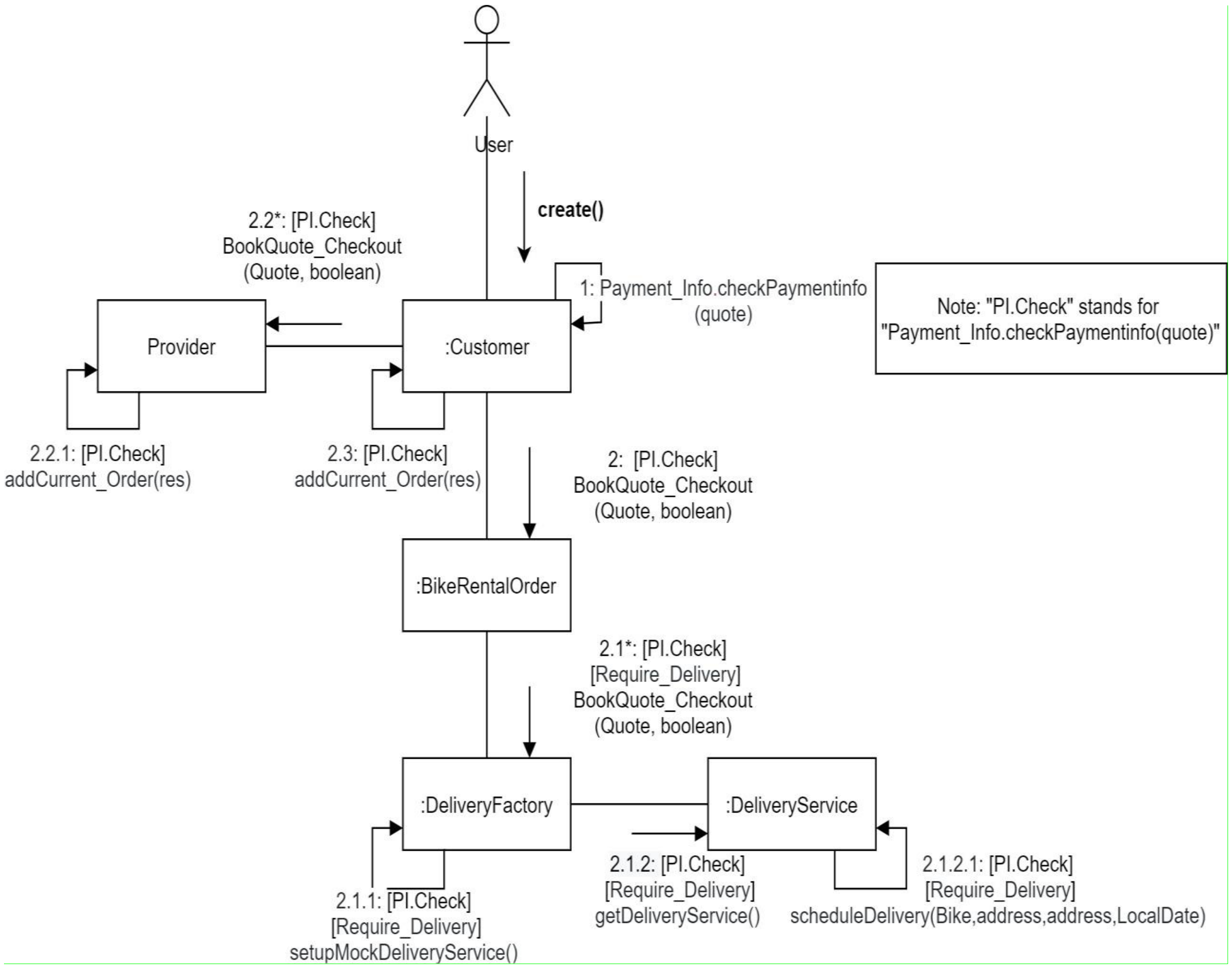

:Customer
:Provider Management System
:Provider
:Request
Bikes of same type :ArrayList<Bike>
:Bike
QuoteRequest (Request, ProviderManagementSystem)
loop
romProvider(Request)
[For all provider in ProviderManagementSystem]
getLocation_of_hire(Request)
L = getLocation_of_hire()
IsNearTo(L)
GenerateQuote(Request)
loop
[For all Bike_Types of requested Bike_Type]
get_BikeTypesFrom(Request)
bts = get_BikeType()
get_Bikes_of_same_type(bt : bts)
loop
[For all Bikes]
get()
Request.getDateRange()
return dr
CheckAvailability(dr)
alt
if[not found]
Expand(dr)
CheckAvailability(dr)
Bike
Quote
Collection<Quote>

Changes made:

1. Deleted unnecessary loops and alternatives options.
2. Deleted many unused classes such as Server, Quote.
3. Add in classes that are necessary for the use case such as ArrayList<Bike>.
4. Add in new methods that are used during the use case.
5. Keep the classes and methods consistent with class diagram and codes.

Communication Diagram

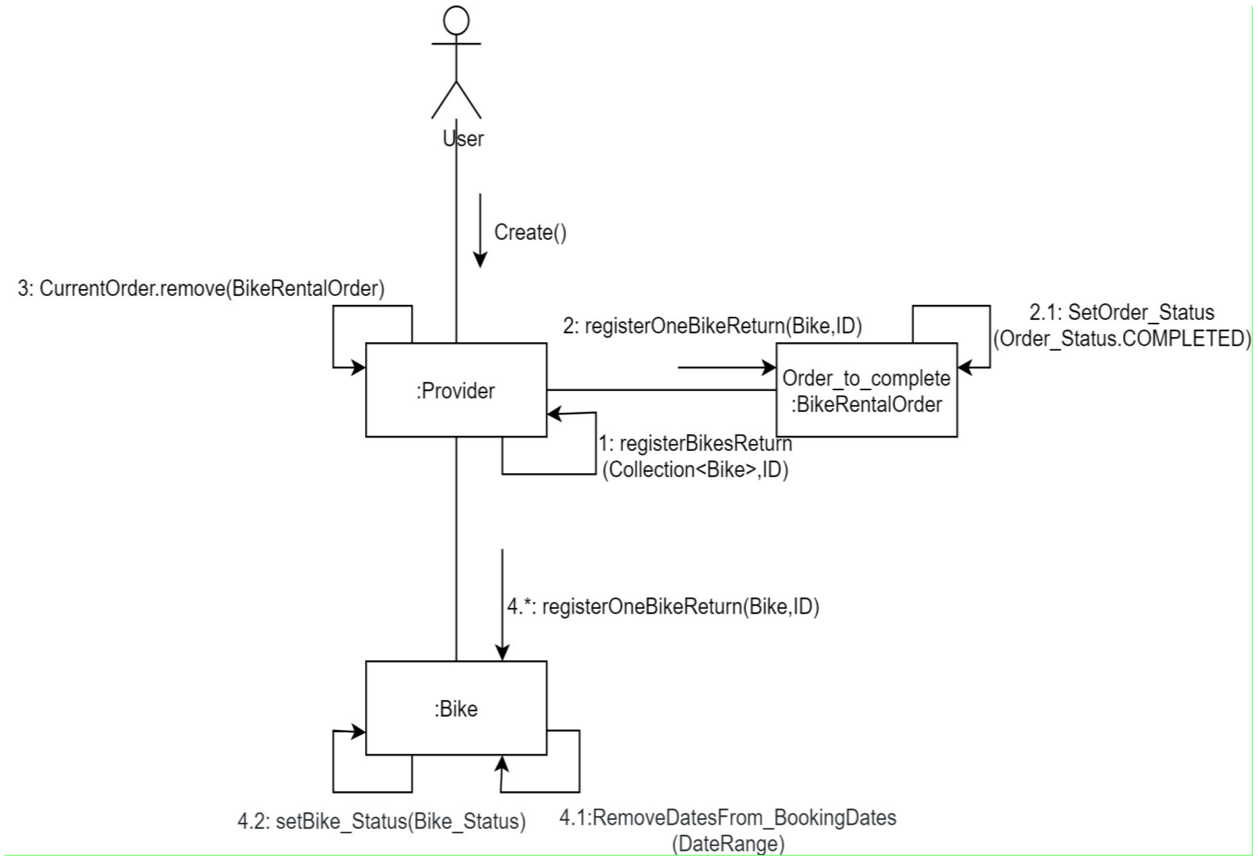
Book Quote:


User
create()
1: Payment_Info.checkPaymentinfo (quote)
Note: "PI.Check" stands for "Payment_Info.checkPaymentinfo(quote)"
2.2*: [PI.Check] BookQuote_Checkout (Quote, boolean)
Provider
:Customer
2.2.1: [PI.Check] addCurrent_Order(res)
2.3: [PI.Check] addCurrent_Order(res)
2: [PI.Check] BookQuote_Checkout (Quote, boolean)
:BikeRentalOrder
2.1*: [PI.Check] [Require_Delivery] BookQuote_Checkout (Quote, boolean)
:DeliveryFactory
:DeliveryService
2.1.1: [PI.Check] [Require_Delivery] setupMockDeliveryService()
2.1.2: [PI.Check] [Require_Delivery] getDeliveryService()
2.1.2.1: [PI.Check] [Require_Delivery] scheduleDelivery(Bike,address,address,LocalDate)

Changes Made:

1. Deleted unused classes and methods.
2. Switch to a more appropriate way of naming and numbering
3. Updated methods and classes based on the latest class diagram.

Register bike return to original provider:


User
Create()
3: CurrentOrder.remove(BikeRentalOrder)
2: registerOneBikeReturn(Bike,ID)
2.1: SetOrder_Status
(Order_Status.COMPLETED)
:Provider
Order_to_complete
:BikeRentalOrder
1: registerBikesReturn
(Collection<Bike>,ID)
4.*: registerOneBikeReturn(Bike,ID)
:Bike
4.2: setBike_Status(Bike_Status)
4.1:RemoveDatesFrom_BookingDates
(DateRange)

Changes made:

1. Taken out Server as a class to keep consistent with class diagram.
2. Switch to a more appropriate way of naming and numbering.
3. Use appropriate arrows to stick with the requirements.

Conformance to requirements:

Two sections in the first coursework are altered to keep consistency with our static and dynamic models. They are respectively **3.2 System State** and **3.3 Use cases**.

System State:

1. After the construction of static and dynamic models, we have found many flaws and rather irrelevant information within our system states. Thus we have edited our system state description based on the attributes of each core classes in our class diagram for consistency.
2. We have also changed the status for bikes and bookings our system has to track. This is because we found some of the status, after we have built up our models, are rather redundant. Thus we edited those to be align with our models.

Use cases:

1. Add relevant stakeholders into the full templates of the first three use cases.
2. Add one more supporting actor which is provider to the use case of ”Get Quote” . This is because in our class diagram, the server is designed to interact with providers in order to access their stocks of bikes.
3. Eliminate anything in use cases related to design step such as “Search button pressed”
4. Add a conditional branch for Use case “Book Quote” to keep consistent with our communication diagram which presents that if the user requires delivery service, the server will schedule a delivery for the customer.

Self-assessment

Criteria	Marks	Justification
Q 2.2.1 Static model	25%	
- Make correct use of UML class diagram notation	4/5%	We have built our class diagram based on the slides within the lecture to ensure the correct usage of notation.
- Split the system design into appropriate classes	4/5%	We have built up 7 core classes and a great number of auxiliary classes. Link to good software practice: Split system into multiple units. Each unit implements one or two tasks of the whole system, resulting in high cohesion.
- Include necessary attributes and methods for use cases	4/5%	We have included attributes and methods into different classes for all our use cases. Yet, we still think that some attributes or methods may be missing as our use cases in cw1 might not be perfect. Link to good software practice: We assigned explicit names to attributes and methods, making our UML diagram more readable.
- Represent associations	4/5%	A wide variety of class

between classes		diagram notations are used, such as inheritance, composition, implementation and dependency.
- Follow good software engineering practices	4/5%	The class diagram follows the rule of low coupling and high cohesion. Multiplicities are added between any two associated classes
Q 2.2.2 High-level description	15%	
- Describe/clarify key components of design	7/10%	We have explained 7 core classes in a high-level perspective.
- Discuss design choices/resolution of ambiguities	3/5%	Three ambiguities in the previous coursework are clarified and resolved in the class diagram.
Q 2.3.1 UML sequence diagram	20%	
- Correctly use of UML sequence diagram notation	5/5%	We used the appropriate notation for sequence diagram. However, some of the notation might be misused.
- Cover class interactions involved in use case	10/10%	With certain change to use case and class diagram, we have discussed and showed all the interaction within our sequence diagram.
- Represent optional, alternative, and iterative behaviour where appropriate	5/5%	We have considered one alternative if the quotes are not found, the system will extend 3 days in date range and search for desired quotes one more time.

Q 2.3.2 UML communication diagram	15%	
- Communication diagram for record bike return to original provider use case	6/8%	We think that our notations might be incorrect such as the numbering in front of each method.
- Communication diagram for book quote use case	7/7%	We think that, based on our class diagram, we have considered all the different interaction between involved classes.
Q 2.4 Conformance to requirements	5%	
- Ensure conformance to requirements and discuss issues	4/5%	All necessary changes are made and explained in detail. But there might still be some gaps between use cases and our dynamic model.
Q 2.5.3 Design extensions	10%	
- Specify interfaces for pricing policies and deposit/valuation policies	2/3%	We have specified two interfaces for pricing and deposit policies. They have specific methods to calculate the prices and deposits. Linked to good software practice: The extension submodules are independent. The system will still operate properly without them, resulting low coupling between the submodules and the system.

- Integrate interfaces into class diagram	5/7%	Interface was added to the class diagram, implementing to relevant classes which needs the abstract method provided by the interface.
Q 2.6 Self-assessment	10%	
- Attempt a reflective self-assessment linked to the assessment criteria	5/5%	We have reflected our project based on the exam assessment criteria.
- Justification of good software engineering practice	4/5%	For each method and attributes, we have stated relevant variable name to give reader a basic understanding about the functionality of it. We also stated the data types of parameters and output for each methods and attributes.