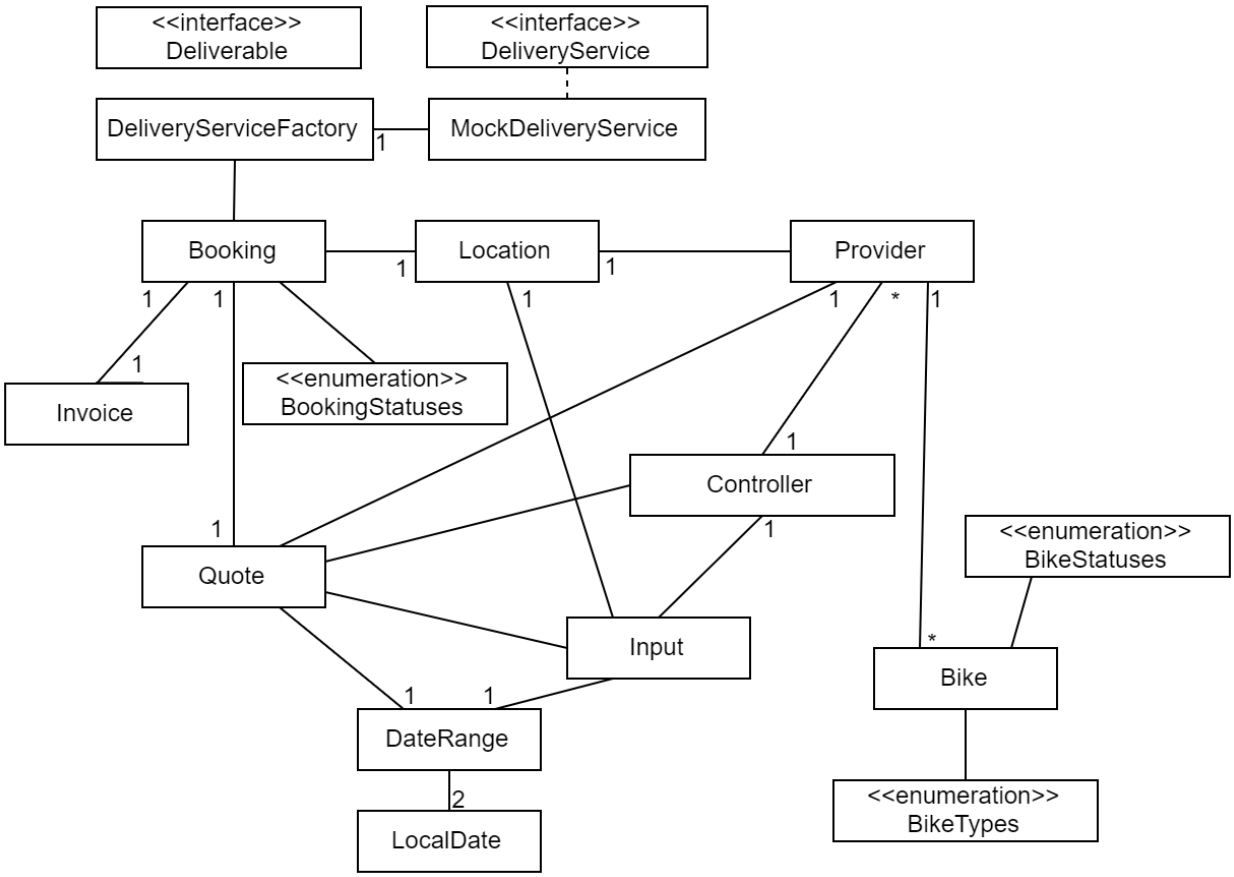
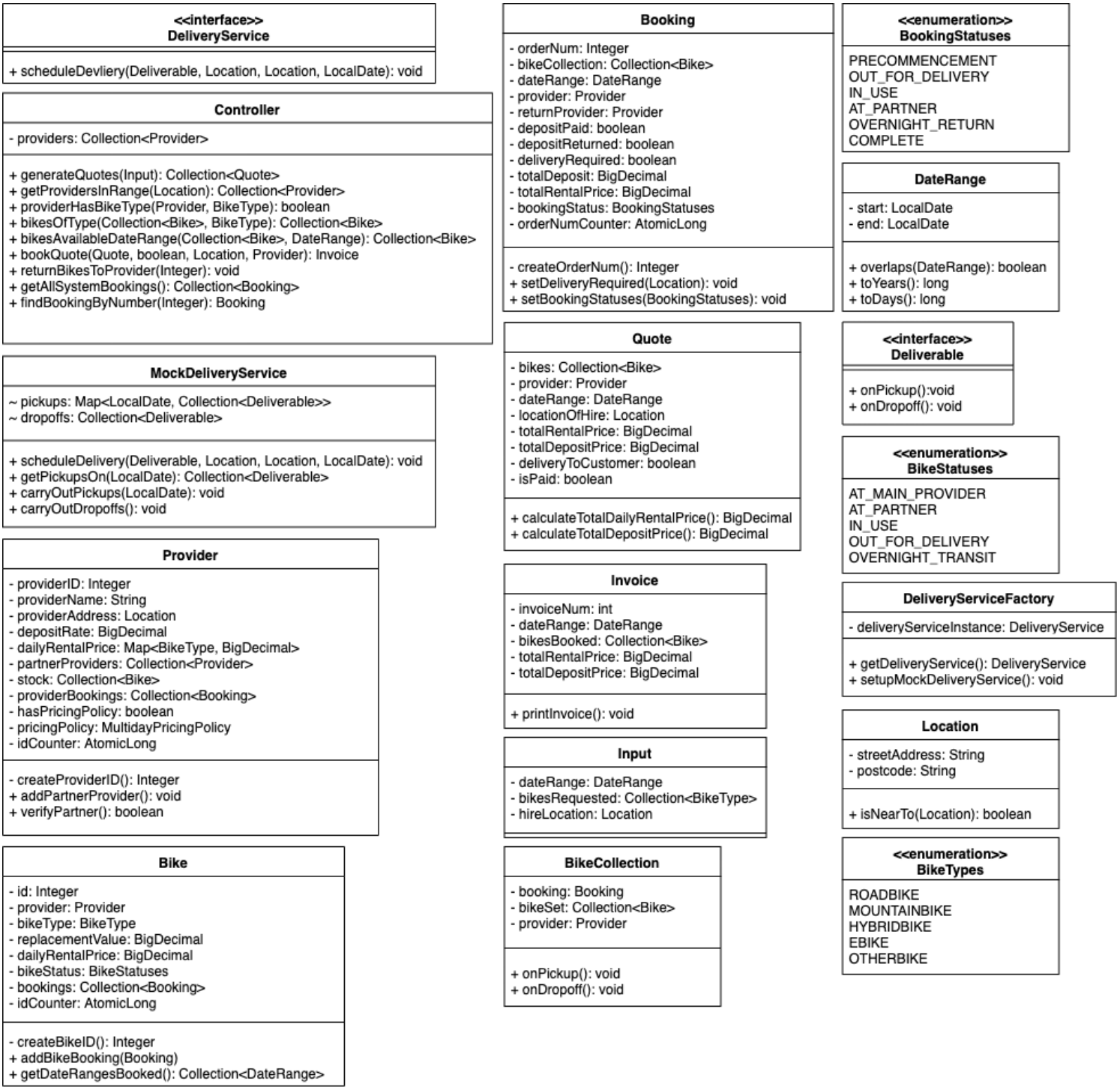
Table of Contents

Question 2.2.1 – UML Class Model (updated)

Entire diagram updated, please mark in its entirety.



(Note: class diagram details on next page)


<<interface>>
DeliveryService
+ scheduleDevliery(Deliverable, Location, Location, LocalDate): void
Controller
- providers: Collection<Provider>
+ generateQuotes(Input): Collection<Quote>
+ getProvidersInRange(Location): Collection<Provider>
+ providerHasBikeType(Provider, BikeType): boolean
+ bikesOfType(Collection<Bike>, BikeType): Collection<Bike>
+ bikesAvailableDateRange(Collection<Bike>, DateRange): Collection<Bike>
+ bookQuote(Quote, boolean, Location, Provider): Invoice
+ returnBikesToProvider(Integer): void
+ getAllSystemBookings(): Collection<Booking>
+ findBookingByNumber(Integer): Booking
MockDeliveryService
~ pickups: Map<LocalDate, Collection<Deliverable>>
~ dropoffs: Collection<Deliverable>
+ scheduleDelivery(Deliverable, Location, Location, LocalDate): void
+ getPickupsOn(LocalDate): Collection<Deliverable>
+ carryOutPickups(LocalDate): void
+ carryOutDropoffs(): void
Provider
- providerID: Integer
- providerName: String
- providerAddress: Location
- depositRate: BigDecimal
- dailyRentalPrice: Map<BikeType, BigDecimal>
- partnerProviders: Collection<Provider>
- stock: Collection<Bike>
- providerBookings: Collection<Booking>
- hasPricingPolicy: boolean
- pricingPolicy: MultidayPricingPolicy
- idCounter: AtomicLong
- createProviderID(): Integer
+ addPartnerProvider(): void
+ verifyPartner(): boolean
Bike
- id: Integer
- provider: Provider
- bikeType: BikeType
- replacementValue: BigDecimal
- dailyRentalPrice: BigDecimal
- bikeStatus: BikeStatuses
- bookings: Collection<Booking>
- idCounter: AtomicLong
- createBikeID(): Integer
+ addBikeBooking(Booking)
+ getDateRangesBooked(): Collection<DateRange>
Booking
- orderNum: Integer
- bikeCollection: Collection<Bike>
- dateRange: DateRange
- provider: Provider
- returnProvider: Provider
- depositPaid: boolean
- depositReturned: boolean
- deliveryRequired: boolean
- totalDeposit: BigDecimal
- totalRentalPrice: BigDecimal
- bookingStatus: BookingStatuses
- orderNumCounter: AtomicLong
- createOrderNum(): Integer
+ setDeliveryRequired(Location): void
+ setBookingStatuses(BookingStatuses): void
Quote
- bikes: Collection<Bike>
- provider: Provider
- dateRange: DateRange
- locationOfHire: Location
- totalRentalPrice: BigDecimal
- totalDepositPrice: BigDecimal
- deliveryToCustomer: boolean
- isPaid: boolean
+ calculateTotalDailyRentalPrice(): BigDecimal
+ calculateTotalDepositPrice(): BigDecimal
Invoice
- invoiceNum: int
- dateRange: DateRange
- bikesBooked: Collection<Bike>
- totalRentalPrice: BigDecimal
- totalDepositPrice: BigDecimal
+ printInvoice(): void
Input
- dateRange: DateRange
- bikesRequested: Collection<BikeType>
- hireLocation: Location
BikeCollection
- booking: Booking
- bikeSet: Collection<Bike>
- provider: Provider
+ onPickup(): void
+ onDropoff(): void
<<enumeration>>
BookingStatuses
PRECOMMENCEMENT
OUT_FOR_DELIVERY
IN_USE
AT_PARTNER
OVERNIGHT_RETURN
COMPLETE
DateRange
- start: LocalDate
- end: LocalDate
+ overlaps(DateRange): boolean
+ toYears(): long
+ toDays(): long
<<interface>>
Deliverable
+ onPickup():void
+ onDropoff(): void
<<enumeration>>
BikeStatuses
AT_MAIN_PROVIDER
AT_PARTNER
IN_USE
OUT_FOR_DELIVERY
OVERNIGHT_TRANSIT
DeliveryServiceFactory
- deliveryServiceInstance: DeliveryService
+ getDeliveryService(): DeliveryService
+ setupMockDeliveryService(): void
Location
- streetAddress: String
- postcode: String
+ isNearTo(Location): boolean
<<enumeration>>
BikeTypes
ROADBIKE
MOUNTAINBIKE
HYBRIDBIKE
EBIKE
OTHERBIKE

Question 2.2.2 – High-level Description

Our design makes use of a central controller class ~~System~~ `Controller`. It has been designed to control all the other classes by having methods that instantiate all the objects in our system.

In the previous coursework we decided unique ID attributes would be useful for bikes and providers, and that decision has been carried through into this stage of the design.

We initially considered whether a kind of sub-booking object would be useful, for the intermediate stage where a user has selected a quote, and then added the required information (personal information, and their mode of collecting the bikes). This information would have to be stored somewhere before the user has made the payment and finalized the booking. We therefore chose to instantiate a `Booking` object by calling the method ~~`generateBooking(Quote)`~~ `bookQuote(Quote)` that takes the chosen quote as an argument, only when a payment has been made. This is because creating a booking without having ensured that payment has been made does not seem logical. And as such, our ~~`generateInvoice(Booking)`~~ `Invoice` constructor ensures that an invoice cannot be created without a booking having been made and thus without having received payment either.

For a few methods, we have made the return type a boolean that will represent whether the method was carried out correctly. For example `extendDateRange()` (not included on class diagram due to the method not being required) would return true if an extended date range object was successfully created, this is to provide a way of knowing whether a task was carried out successfully. This might be redundant at a lower level, in which case the return value would just be void.

A major confusion was the `PricingInterface` class in that the arguments these methods would need to take in was very ambiguous. In theory the provider could decide to calculate the daily rental price and the deposit amount in any way they wanted, and as a result until we know how they want the method to work, we cannot be sure what arguments it would need. We have however just assumed that they would be the default types.

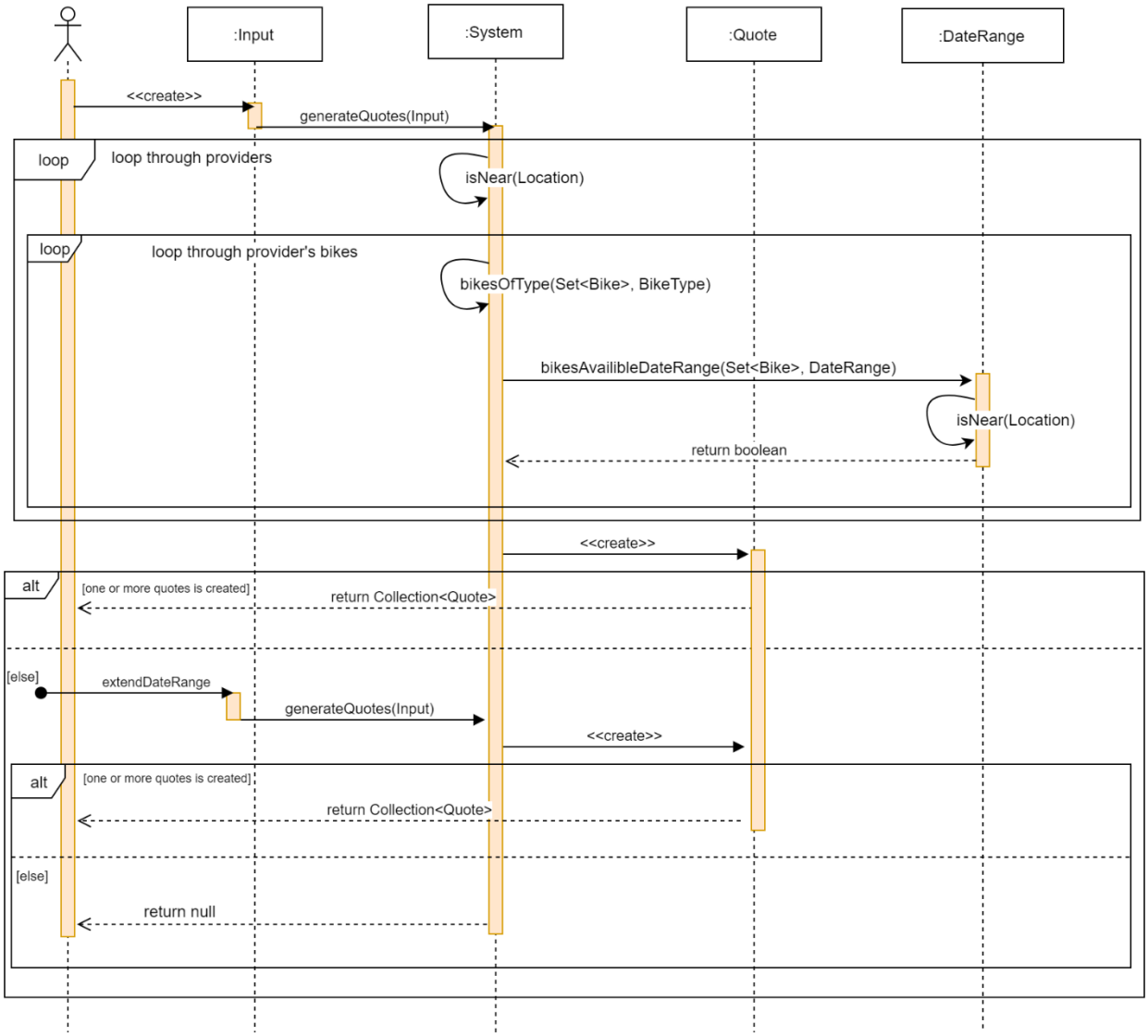
In the brief, it was mentioned to leave out details of `DeliveryService`. This leads to ambiguity regarding what the class represents, and subsequently its relationship and associations with other classes. We have chosen to interpret that an instance of the `DeliveryService` class represents a single trip to a destination, hence its multiplicity of 0..2 in the UML class diagram. Each booking can have no deliveries (if the bikes are collected and dropped off by the customer at the main provider), one delivery (either to the customer, or from a partner provider to the main provider) or two (both the previous two options).

There were several options for how and when to update the statuses of bookings and bikes, specifically when they were being returned. One option was to update the status of the bikes as they were returned, and for the booking status to be cleared automatically when all the bikes included were returned. However, we have decided that it makes more sense for the status of the booking to be explicitly updated rather than implicitly through the individual bike statuses. That is that by calling the ~~`updateBookingStatus(BookingStatuses)`~~

`setBookingStatus(BookingStatuses)` method, the ~~`updateStatus(BikeStatuses)`~~ `setBikeStatus(BikeStatuses)` method will be called for every instance of bike involved in the booking. However, this is a potential area for revision during the implementation.

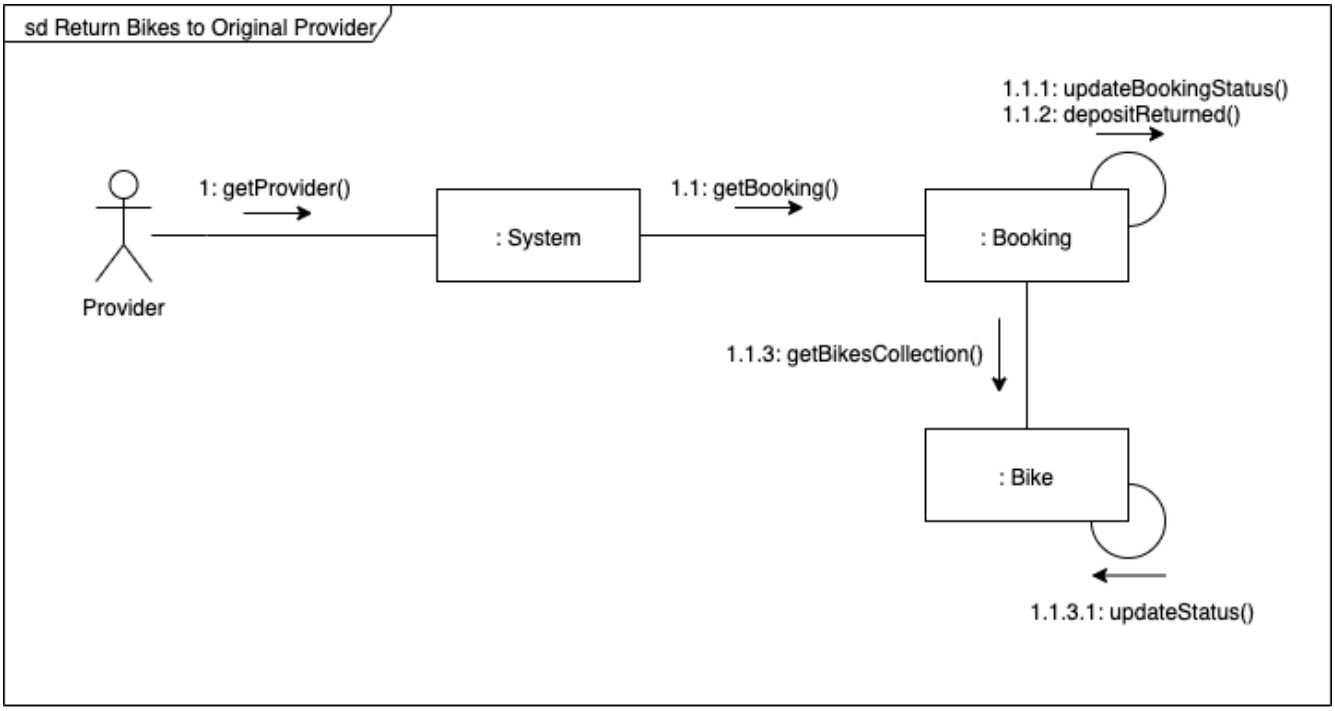
Question 2.3.1 – UML Sequence Diagram (updated)

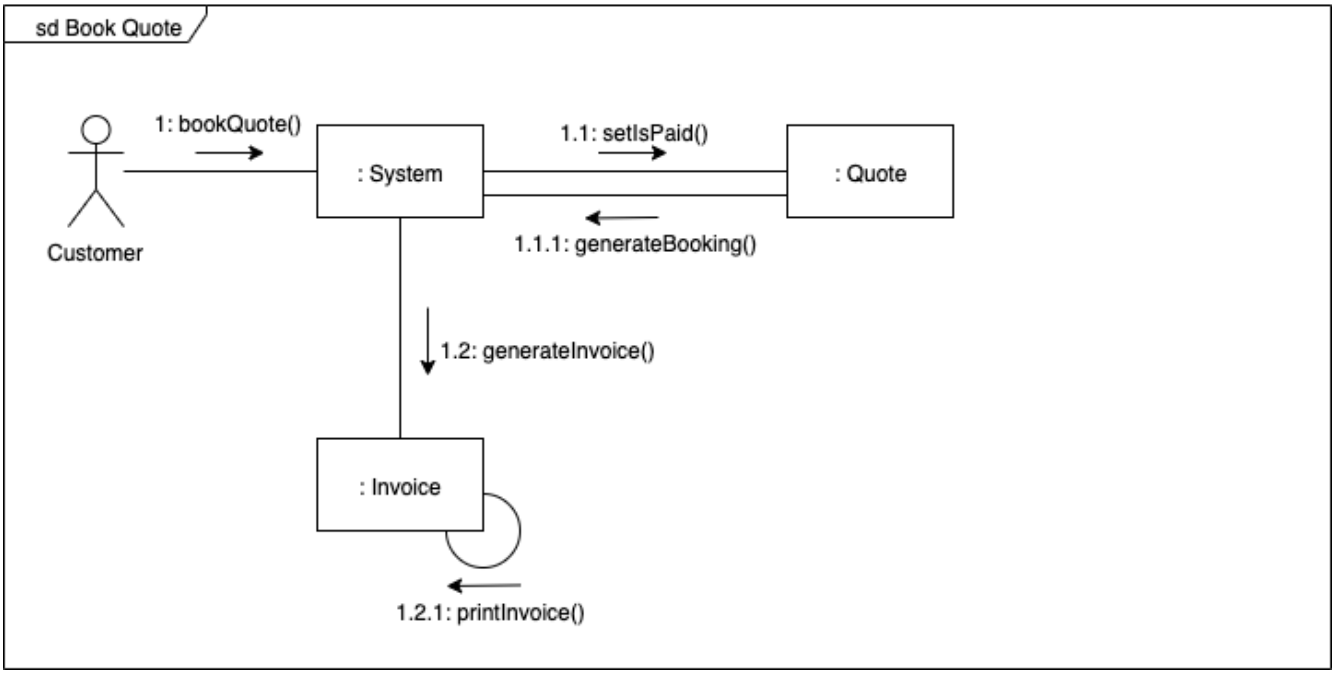
Entire diagram updated, please mark in its entirety.



Question 2.3.2 – UML Communication Diagram

For conciseness, method arguments have been omitted.





Question 2.4 – Conformance to Requirements

Originally we thought to keep personal information for returning customers, and therefore require that they log in to the system with a username and password. However this unnecessarily complicates the design and wasn't specifically requested, nor is it particularly necessary at the moment. Should the stakeholders decide at a later stage that they would like this feature to be added to the system, it would not be that hard to implement.

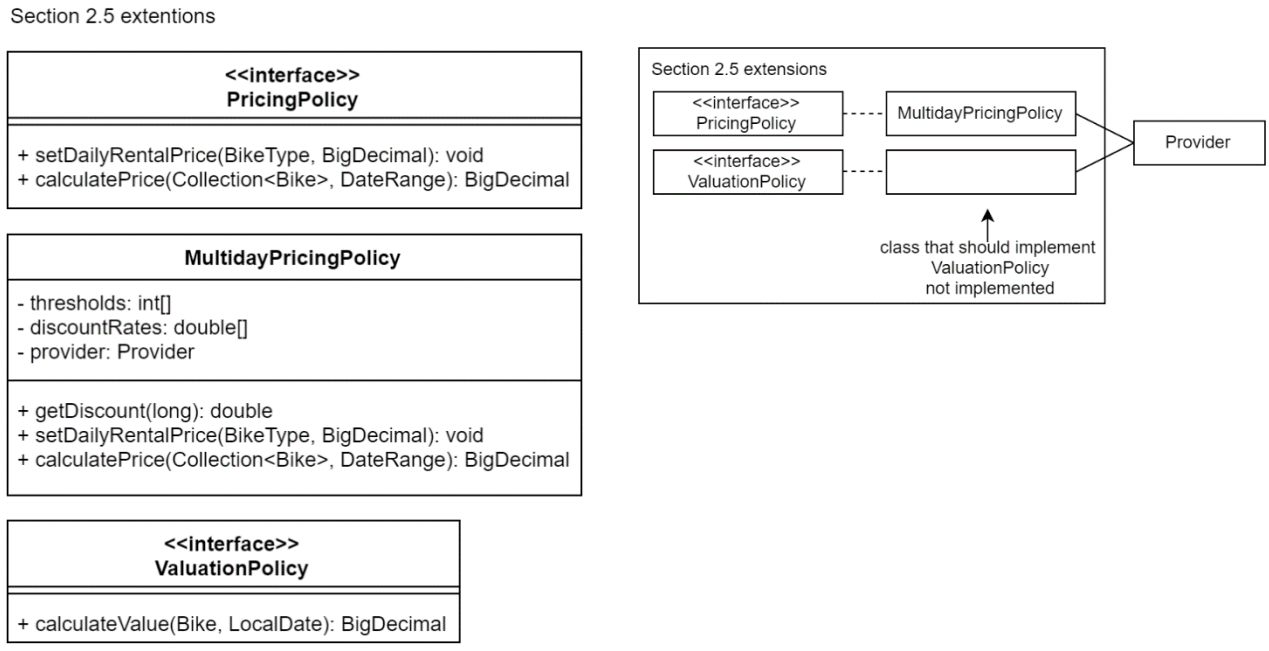
As discussed earlier, the *Book quote* use case has changed due to the system no longer being able to identify that the current customer is a regular, and thus no information will be saved for them.

Other than the above-mentioned changes, all other requirements specified in coursework 1 have been taken into consideration for the design of the system.

Quote state added to coursework 1.

Question 2.5.3 – Design Extensions

The client's desire to make the pricing policies extendable was recognised in the form of an interface called ~~`PricingInterface`~~ `PricingPolicy`, which is implemented by a ~~`PricingPolicy`~~ `MultiDayPricingPolicy` class. In turn, each provider will have an association with a custom ~~`PricingPolicy`~~ `MultiDayPricingPolicy` object as shown below. This allows for the hired external contractors to create custom pricing and custom deposit rate policies. The `PricingInterface` has default methods for calculating prices as described by coursework 1, but these can be easily modified in the `PricingPolicy` class to suit the needs of the providers, including the ability to take depreciation into account and other factors that may affect the valuation of the bikes.



Question 2.6 – Self-assessment

Q 2.2.1 – UML Class Model (21/25)

- Make correct use of UML class diagram (5/5)
  - UML syntax is used correctly to describe the classes and their multiplicities with each other.
- Split the system design into appropriate classes (5/5)
  - All classes that are required in our design of the system have been represented.
- Include necessary attributes and methods for the use cases (4/5)
  - All the methods and attributes we expect to need are listed, however there might be some that we need but have missed.
- Represent association between classes (4/5)
  - Accurate associations have been represented, however again some might have been missed.
- Follow good software engineering practices (3/5)
  - The coupling present may not be ideal as the modules are somewhat interdependent and share global data structures. However, the modules are quite cohesive as they are based on real objects, resulting in logical attributes and methods that represent the module.

Q 2.2.2 – High-level Description (15/15)

- Describe / clarify key components of design (10/10)
  - All classes, methods and relations between them are explained in further detail if we thought it was needed.
- Discuss design choices / resolution of ambiguities (5/5)
  - Where we carefully considered other options, the motivation behind our decision is summarised to explain why we made the choice we did.
  - Ambiguities in the system requirements was also discussed as well as the assumptions we made when we could not entirely understand the requirements as they were laid out. Further discussion with the stakeholders would be required at this stage to clarify whether the design decisions we have made are really what reflective of the system they envisaged.

Q 2.3.1 – UML Sequence Diagram (18/20)

- Correct use of UML sequence diagram notation (5/5)
  - UML sequence diagram notation used properly and in full.
- Cover class interactions involved in use case (8/10)
  - The class interactions that we supposed would take place have been covered. However it is noted that the diagram might be too simple, and that essential extra steps might not have been realised. Overall, the most obvious steps have been shown.

- Represent optional, alternative, and iterative behaviour where appropriate (5/5)
    - Alternative frames have been used correctly, labelled with their success and else cases.

Q 2.3.2 – UML Communication Diagram (14/15)

- Communication diagram for return bike to the original provider use case (7/8)
    - The correct syntax is used, and the communications are consistent with the class diagrams. Getter and setter methods mentioned were not explicitly shown in the class diagram, but were implied for all attributes.
- Communication diagram for book quote use case (7/7)
    - As above, the correct syntax is used and the communications and messages are consistent with the class diagram.

Q 2.4 – Conformance to Requirements (5/5)

- Ensure conformance to requirements and discuss issues (5/5)
    - Changes to the requirements in coursework 1 that have been made have been discussed, including how the changes have impacted our design here.

Q 2.5.3 – Design Extensions (7/10)

- Specify interfaces for pricing policies and deposit / valuation policies (2/3)
    - In interface with two methods was created, as well as a class that would implement this interface. As discussed above, the arguments these methods should take was unclear to us, and thus may not be correct.
- Integrate interfaces into class diagrams (5/7)
    - Interface and implementation class integrated into the design to some extent, however we are uncertain this is how it is meant to be done.

Q 2.6 – Self-assessment (9/10)

- Attempt a reflective self-assessment linked to the assessment criteria (5/5)
    - Each section was reflected upon, and an honest and thorough evaluation was given. For each section a reasonable mark was indicated reflective of the written assessment we gave ourselves.
- Justification of good software engineering practice (4/5)
    - Our high-level design meets all the functional requirements (it is however not clear at this stage whether all the non-functional requirements will be met, but where possible, they have been taken into consideration in the design process). Our design allows is easily extendable, and make use of existing Java classes like `LocalDate` and `Currency` where necessary.