INF2C SOFTWARE ENGINEERING 2019-20 COURSEWORK 2

Capturing requirements for a federalized bike rental system

- HUACHENG SONG s1826390
- XUDONG ZHANG s1817972

Q2.1 Introduction:

This document illustrates the design of the rental bike system with static model and dynamics model. After reading this document, you could know how this system words without any ambiguous idea.
For further information on requirements, please refer to: Coursework 1 Instructions.

Q2.2 Static model:

- 2.2.1 UML class model:

Figure 1:

Show just class names and the associations and other dependencies such as generalization dependencies between classes.

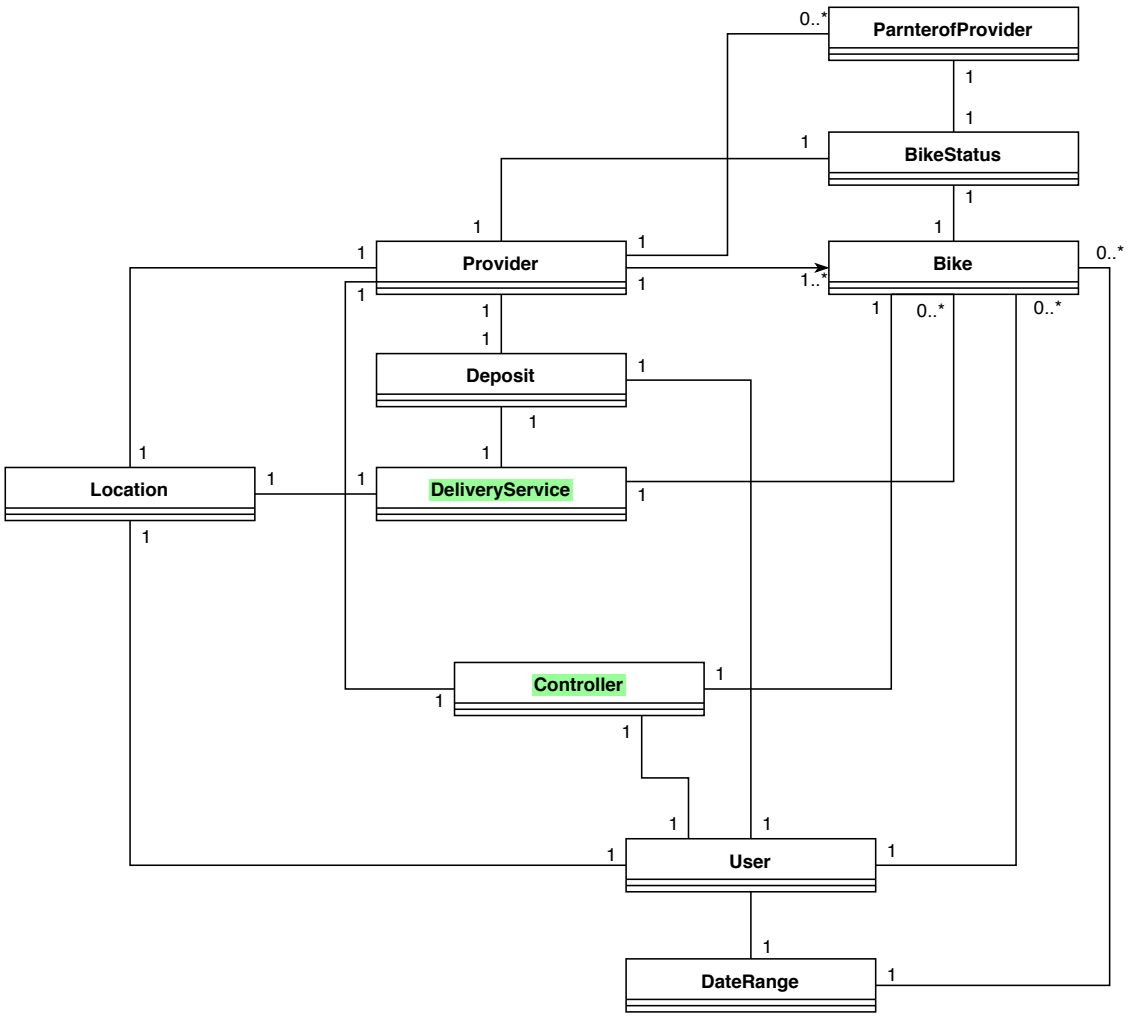
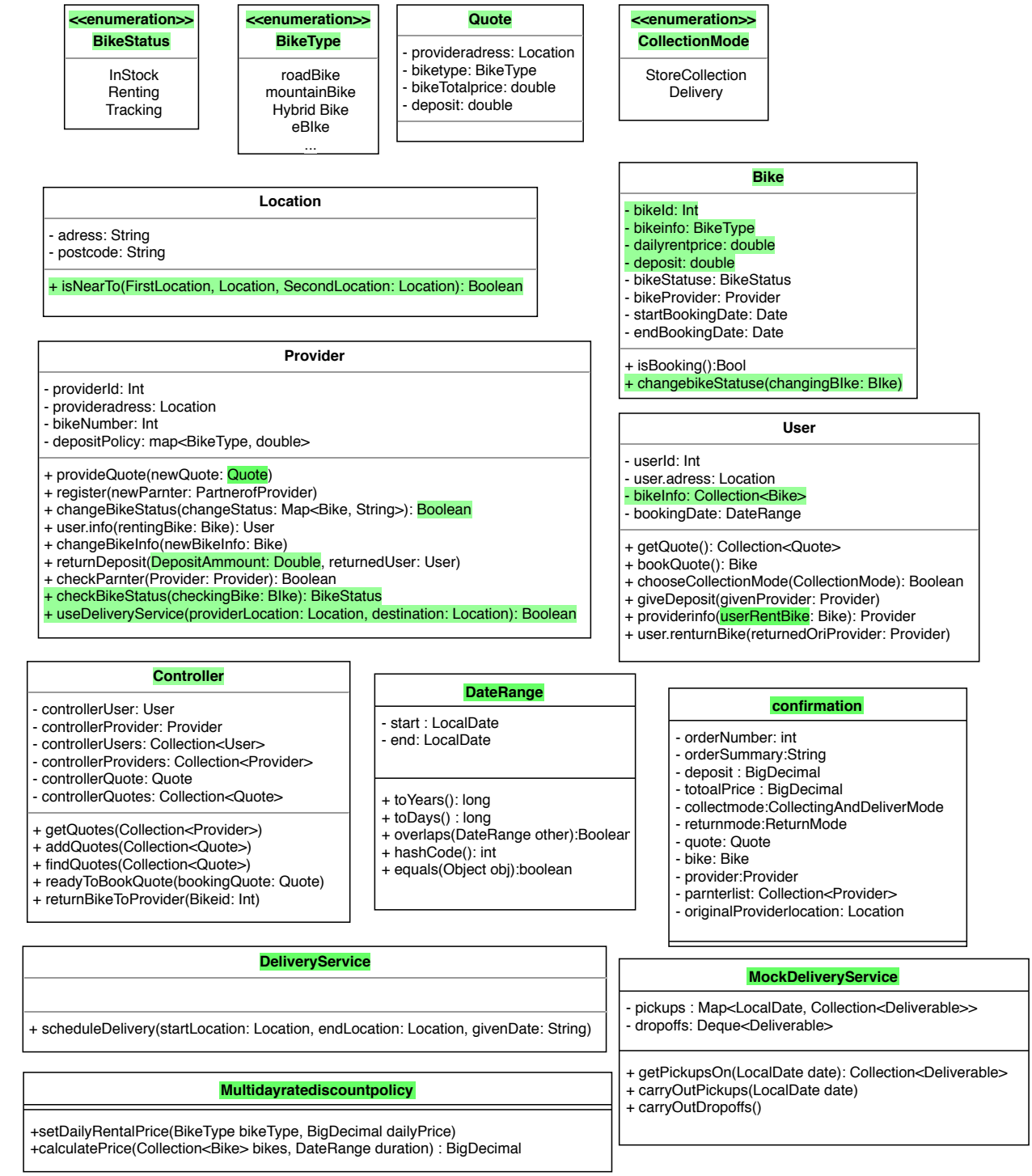


Figure 2:

Omit the associations, show for each class its attributes and operations.



- 2.2.2 High-level description:

1. Justification for the design:

1) Methods to increase cohesion and reduce coupling:
Most of the functions in each class only take its own class variables as arguments or act as a setter of the class variables which increases cohesion.
e.g.:
providerQuote(newQuote: Bike)

getQuote(): Collection<Bike>
bookQuote(): Bike
giveDeposit(givenProvider: Provider)
changeBikeStatus(changeStatus: Map<Bike, BikeStatus>)
As the provider provide the information of Bikes, user will use getQuote() to get the collection of Bikes.

2) Encapsulation and information hiding in the design:
For all classes in the UML class diagram, variables are declared as private variables (e.g. in class Provider variable "bikeNumber", "postcode" and "address" are all declared as private variables) and can only be accessed via getter and setter so that they are hidden from other classes.

2. Assumptions concerning ambiguities and missing information:

1) Unregistered users might could not even able to look though the provided bikes.
2) As no transfer information noted hence, we assume the transfer method will automatically calculate but forget to add the specific method.
3) Lacking method to allow the partner of provider call delivery drivers in order to return bikes to original providers.

3. Explanatory notes of class diagram:

1) Class "BikeStatus":
BikeStatus class contains no methods but three private attributes RentingStatus, AvaliableStatus and TrackingStatus. These attributes show the present state of the bike.

2) Class "Location":
Location class contains no methods, but two private attributes address and postcode. These two attributes will clarify the location of provider.

3) Class "Deposit":
Deposit class contains one private attribute and four functions. "inUser()", "inProvider()", "inParnter()", "indriver()" have no parameters and have return Boolean value in order to check whose hand holds the deposit.

4) Class "Bike":
Bike class contains six attributes: bikeType, bikeStatuse, bikeRentprice, bikeProvider, requiredDate, bookingUser and these will uniquely identify bikes. One method, "isBooking()" will check whether the bike is booked.

5) Class "Daterange":
Daterange contains two attributes startDate and endDate and contains one method "RequireToReturn()".

6) Class "Provider":
Provider class contains five attributes:location, bikeNumber, bikeStatus, depositStatus, haveParnter. And also contains six methods, "provideQuote(newQuote: Bike)" will provide the new bike information to the system, "register(newParnter: PartnerofProvider)" will register with the other provider and then they become each partner, "changeBikeStatus(changeStatus: Map<Bike, BikeStatus>)" will change the BikeStatus in a Mashmap., "userInfo(p.rentingBike: Bike): User" will get the information of user who is renting the parameter bike, "changeBikeInfo(newBikeInfo: Bike)" and "returnDeposit(returnedUser: User)".

7) Class "ParnterofProvider" :

ParnterofProvider is a subclass of Provider. It contains two additional methods, Register and useDeliverydriver.

8) Class "User" :

User class contains five attributes: location, rentStatus, orderNumber, bikeInfo and bookingDate. Ordernumber is unique for each booking. And the class contains six methods. "getQuote()" will receive collection of Bikes and "bookQuote()" will book a bike hence the return value is Bike.

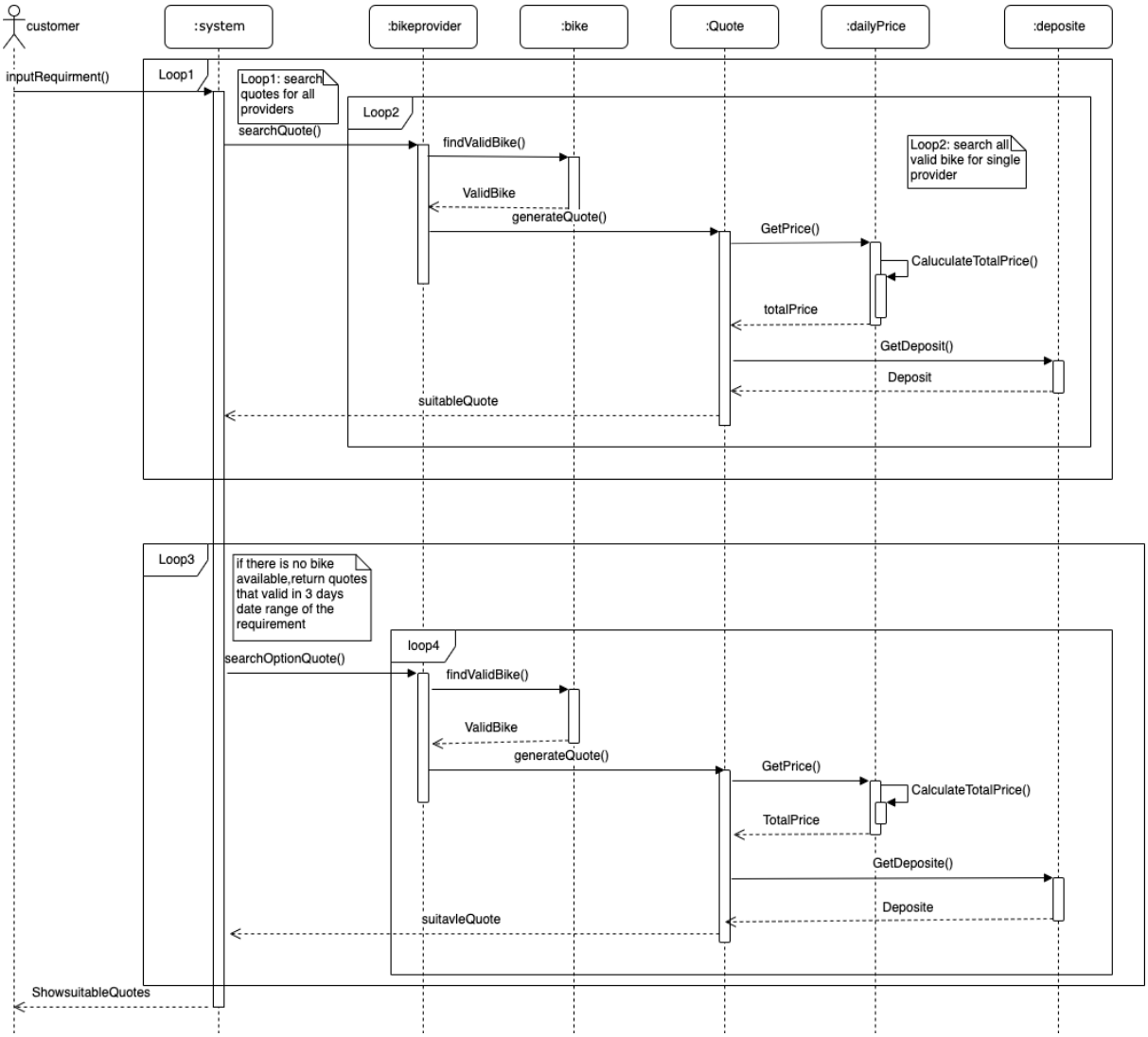
9) Class "Deliverydriver" :

Deliverydriver class contains two attributes TrackState and depositStatus and three methods, register, record and driver.returnBike.

Q2.3 Dynamic models:

● 2.3.1 UML sequence diagram:

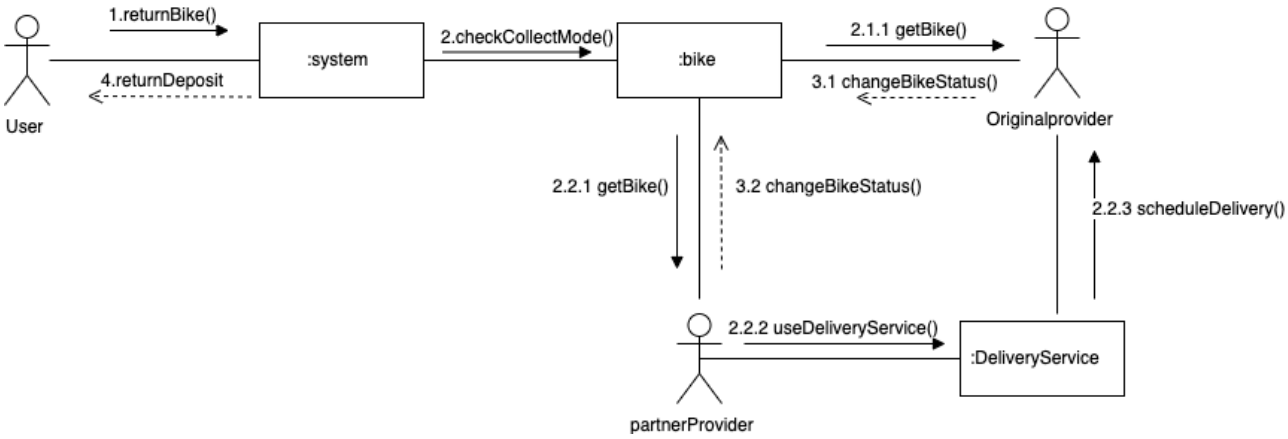
A UML sequence diagram for the Get Quotes use case.

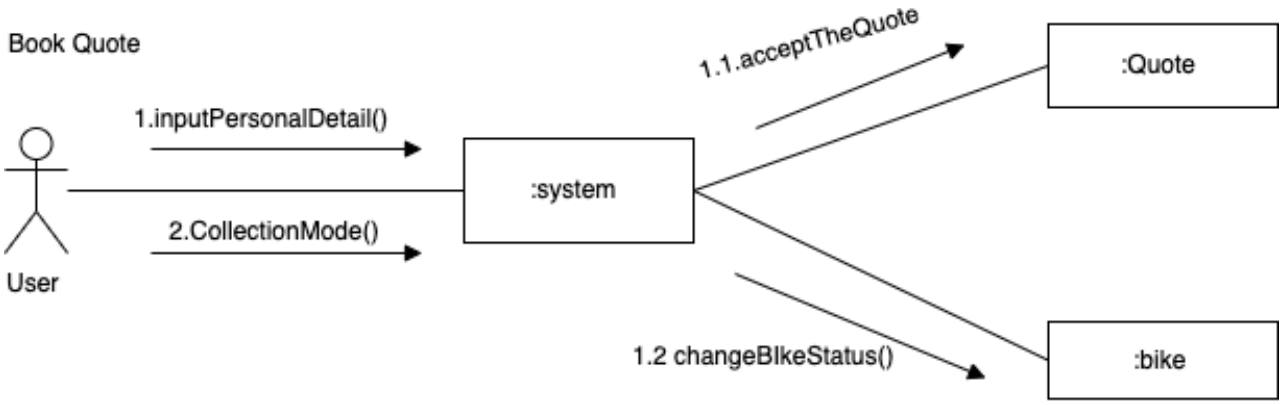


- 2.3.2 UML communication diagram:

communication diagrams (also sometimes called collaboration diagrams) can be used to focus on the organization of the objects and their interaction.

Return Bike to originalProvider





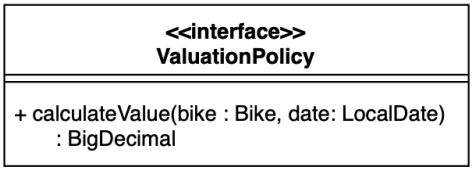
Q2.4 Conformance to requirements:

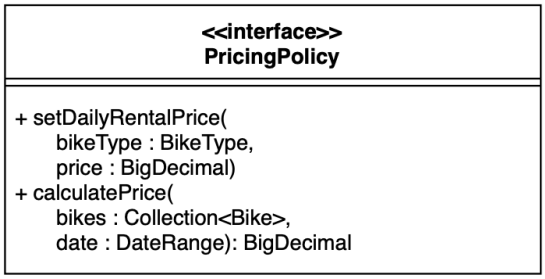
Apparent divergences:

- We add one more actor “Delivery drivers” .
- We add more use cases to make the system more complexity.

Q2.5 Design extensions:

- 2.5.3 Modified design:





Q2.6 Self-assessment:

Attempt a reflective self-assessment linked to the assessment criteria

- Q 2.2.1 Static model 25%
  Make correct use of UML class diagram notation 5%
  We use UML class diagram notation correctly (show by the graph).
  Split the system design into appropriate classes 5%
  We split design correctly (show by the graph).
  Include necessary attributes and methods for use cases 5%
  We add correct attributes for the specified class as the question asked (show by the graph).
  Represent associations between classes 5%
  We represent correct associations for the specified class as the question asked (show by the graph).
  Follow good software engineering practices 5%
  We use simple design and did pair working to make work simpler.
- Q 2.2.2 High-level description 15%
  Describe/clarify key components of design 10%
  We clarify and describe how the key components work in the class and also the attributes and methods in the classes.
  Discuss design choices/resolution of ambiguities 5%
  We discuss the ambiguities e.g.: some customers might have no account so they could not even look through the quotes.
- Q 2.3.1 UML sequence diagram 20%
  Correctly use of UML sequence diagram notation 5%
  We correctly draw the classes after group discussion.
  Cover class interactions involved in use case 10%
  Represent optional, alternative, and iterative behavior where appropriate 5%
  We used alternative also we used loop.
- Q 2.3.2 UML communication diagram 15%
  Communication diagram for return bikes to original provider use case 8%
  Communication diagram for book quote use case 7%

We showed the process of booking a quote in the book quote communication diagram and showed the communication between the User, Controller, Provider and the Bike class.

- Q 2.4 Conformance to requirements 5%
  Ensure conformance to requirements and discuss issues 5%
- Q 2.5.3 Design extensions 10%
  Specify interfaces for pricing policies and deposit/valuation policies 3%
  We add extra methods to illustrate the pricing policies(as on the interface class).
  Integrate interfaces into class diagram 7%
  Show on the graph.
- Q 2.6 Self-assessment 10%
  Attempt a reflective self-assessment linked to the assessment criteria 5%
  Justification of good software engineering practice 5%

Justification of good software engineering practice:

In our rental system, most of system function use private variables to work, this could avoid any chaos when different functions call the same kind of variables. As well as, each function in our system have clear target, so system would not waste any times to deal with unnecessary step. Two interface pricing policies and deposit policies are designed in this system, this could increase the availability of the system and make renting more rational. Our system contains more specific use case (get Quote, book Quote, update the bike information ...), so that this system could cover all requirements for renting bikes. In coursework2, we realized that the requirement would change from cw1 to cw2, so we made some important change to cover those requirements, like the class of controller and interface of pricing-deposit polices……Furthermore, we use teamwork to solve this coursework, so we could finished task faster and simpler. We will share our opinions when we are struggle, we could understand each other' s part job very quickly, so we are confidence that other software engineer could understand our design.
In conclusion, we trust we really did a good job.