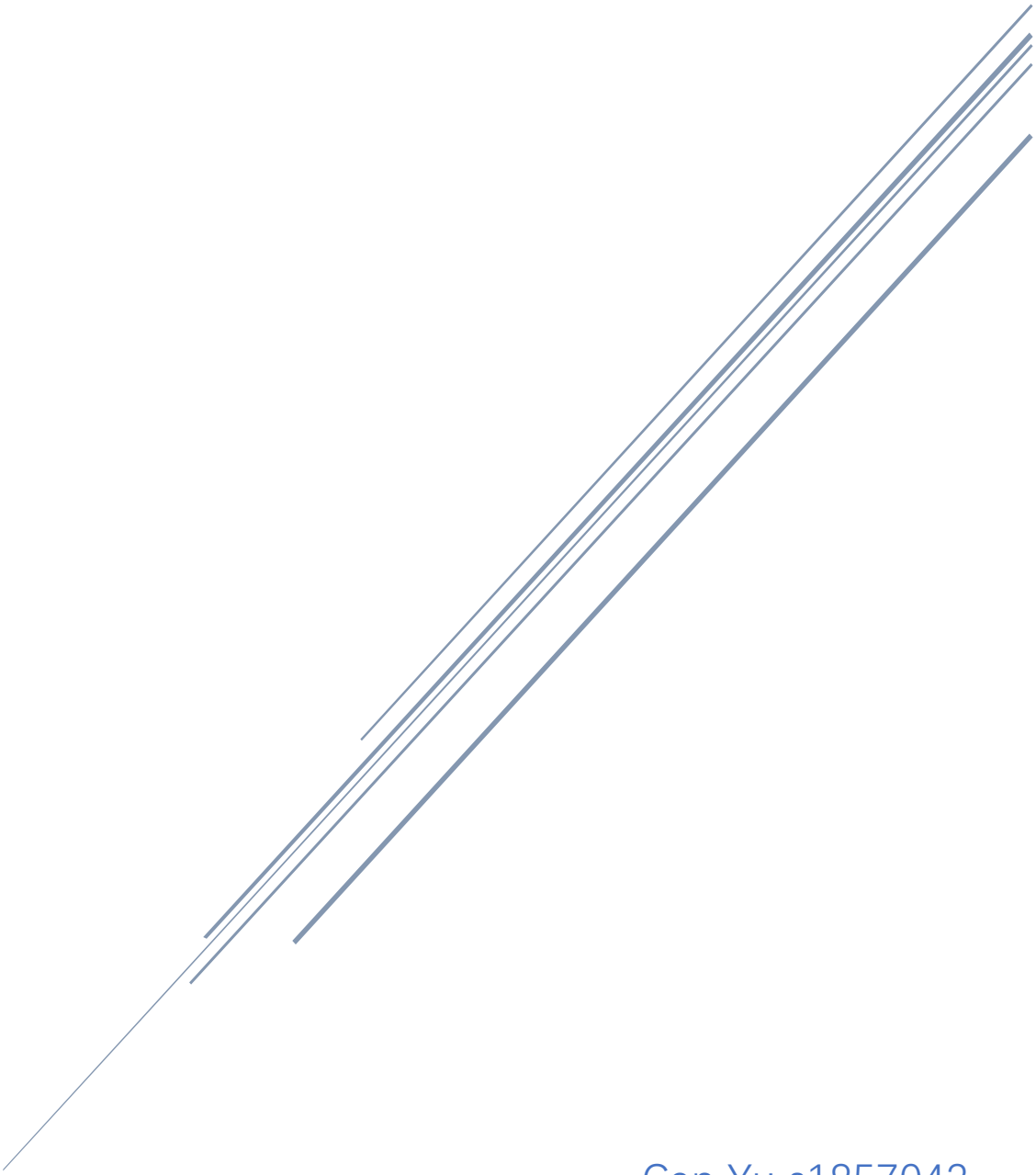
SOFTWARE ENGINEERING

Coursework 3 Report



Cen Yu s1857043
Zhou Shengyi s1864509

Table Of Content

Revision to Requirement

System state:

we have updated all the information that the system needs to keep on customers, providers, bi kes and bookings according to the UML class diagram and our production code.

For Customer & Provider:

1. Replace name, Phone No. and Email address with PersonalInfo, consisting with the desi gn and code.

For Provider:

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 st atus in use case of recording bike return.

For bike:

5. Add Date_of_Purchase to help provider to estimate the replacement value of bikes base d on its age.
6. Add BookedDates to show a bike’s availability in a long period of time.

For book rental order:

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 t o both provider and customer.
8. Add TotalDeposit to indicate how much deposit that customer have to pay before getti ng the bike(s) or how much provider /partner has to return when the customer returns t he bike(s);

For other information the system needs to record:

9. ProviderManagementSystem, keeps all providers' information
10. personalInfo, includes name, contact number,EmailAddr
11. Pricing and Valuation Policy for submodule extension
12. BikeType, in addition to name and model, it also stores replacement value and Default_DailyRentalPrice

For status of bikes:

13. Reduced to 4 different status: [DELIVERING], [ARRIVED], [RENTED], [INPARTNERSHOP] to consist with UML class diagram

For booking status:

14. Reduced to status of [BOOKED], [BIKEDELIVERYING],[BIKEARRIVED],[BIKERETURNEDTOPARTNERSHOP], [COMPLETED] to consist with UML class diagram

Revision to design

Changes made to design document and the reasons behind it:

For class diagram, we have made the following changes：

Changes	Reason
Replace BikeType enum with BikeType class	The BikeType class is better for storing and encapsulating multiple types of data such as name, model and replacement value and defaultDailyRentalPrice.
Eliminate trivial methods in classes	Methods such as Password_Verification(), FindHistoryOrders(Order) , read_input(), report_Demage() and etc. are dumped due to their irrelevance to the main use cases.
Dump User and Account class	It is trivial in design stage as we could simply create an account by using ‘new‘ operator to initialise a provider/customer object.

Delete the whole Server class	Initially, we want Server class to play roles of data base and controller so that it can store all the Provider, Customer, and Order objects and manipulate them, like providing relevant data to each classes' methods, generate orders for customers, schedule delivery service and etc. However, it is against the principle of high cohesion in software development since we should not have a class that is capable of doing so many jobs. Thus, we distributed the functionalities of server to different classes. we have created a ProviderManagementSystem class that only helps customer to get potential quotes from all providers. Meanwhile, we have merged the GenerateOrder method from server with BookQuote method in Customer class to make the process of booking quote more straightforward in coding.
Create DefaultValuationCalculator, LinearDepreciaitonCalculator and DoubleDecliningDepreciationCalculator that implement Valuation policy interface. Add two methods(SwitchValuationPolicy and calculateDepositofABike) in Provider class to help calculate deposit.	We try to take advantage of polymorphism by adding one additional member variable, let us say, "ValuationPolicy calculator" into Provider class. The SwitchValuationPolicy(int,BigDecimal) method allows every Provider to choose different policy to determine replacement values of bikes. Every provider can then calculate deposit of bike based on their different bike replacement value and deposit rate but using the same method calculateDepositofABike (Bike,LocalDate).

Dump partnershipAccountBook class	PartnershipAccountBook records the deposit bala nce between two partners. It is necessary, but we eventually think this is not the best way to deal wi th the deposit balance between two partners. Th e best way should be one side paying back depo sit to another side in shortly time after bike retur n. But simulating online transaction is beyond the scope of this course work. So we decide to skip t his part.

For Higher-Level description:

Based on the latest class diagram, we have rewritten the description of the key components in each core class. We also discuss the choices we have made in the construction of class diagram such as what data type we have used to store bikes and what auxiliary classes we have built to make the whole system function in a more reasonable way.

For Sequence and communication diagram:

Since our sequence and communication diagram heavily relied on Server in the first draft, we altered each diagram thoroughly based on our latest class diagram and actual production code that runs without Server class.

For Self-Assessement of Design.pdf:

We have added some reflective self-assessement regarding good software practice.

Self-Assessment

Criteria	Marks	Justification
Q1. Extension submodules	10%	

- Implementation of extension submodule
  - Should implement extension submodule
  - Should include unit tests for extension submodule
	9/10%	We have created three classes (DefaultValuationCalculator, LinearDepreciaitonCalculator and DoubleDecliningDepreciationCalculator) implementing ValuationPolicy.
		They have the same parent class and same methods, supporting the usage of polymorphism in the system.
		
		We have written unit tests for both LinearDepreciaitonCalculator and DoubleDeclining DepreciationCalculator. For each test, we let each class to calculate replacement value for 5 different aged bikes in the same date.

- Peer review of the other group's submodule		
Q2. Tests	35%	

- System tests covering key use cases
  - Should have comments documenting how tests check the use cases are correctly implemented
  - Should cover all key use cases and check they are carrying out the necessary steps
  - Should have some variety of test data
  - Should use **MockDeliveryService**
	18/20%	Our system tests cover 3 key use cases.
		We have written comments in tests to document how tests check the use cases are correctly implemented.
		For each use cases we tests the correctness of every new data generated and changes that have occured in different objects. For example, in use case of get quote, we test how many quotes should be generated by providers (since we already know the request and stock of all providers). Also, we have checked whether every bike from the quote is available in the requested date range and their total deposit and price are calculated correctly.
		
		We try to simulate both successful and failed scenarios to get different variety of test data. For example, in our tests of get quote. one tests that customer gets quote successfully and another one tests that no quote found due to booking date clash.
		
		In our system tests, **MockDeliveryService is used to simulate delivery service.**

- Unit tests for **Location** and **DateRange**	5/5%	We have written unit tests for Location and DateRange
- Systems test including implemented extension to pricing/valuation	5/5%	We tests how Provider class implements Valuation and Pricing policy to calculate correct deposit and price.
- Mock and test pricing/valuation behaviour given other extension (**Challenging**)	5/5%	We have built a MockMultiDayPricingPolicy class to mimic how the extension submodule calculates the prices of a

		number of bikes based on duration of their booking. We have tested MockMultiDayPricingPolicy in PricingPolicyTest.
Q3. Code	**45%**	

- Integration with pricing and valuation policies
  - System should correctly interface with pricing and valuation policies
  - System should correctly implement default pricing/valuation behaviour
	10/10%	We have implemented extension submodule in system by creating three Valuation Calculator classes (DefaultValuationCalculator, LinearDepreciaitonCalculator and DoubleDecliningDepreciationCalculator) implementing ValuationPolicy interface. Provider has ValuationPolicy as object in its member variable field and they can be reset by calling SwitchValuationPolicy method. Given replacement value of bike and deposit rate inside Provider class, the system is capable of calculating out ideal deposit for provider.
		
		In our production code, system could implement default Valuation behaviour by setting the Calculator (a member variable) to DefaultValuationCalculator in either constructor or by calling method SwitchValuationPolicy(0, depreciation rate)

- Functionality and correctness
  - Code should attempt to implement the full functionality of each use case
  - Implementation should be correct, **as evidenced by system tests**
	22/25%	Our code implements the full functionality of three main use cases to a large extent.
		
		Our code has passed our own system testes which is reasonably designed. We think our implementation be correct to a large extent.

- Quality of design and implementation
  - Your implementation should follow a good design and be of high quality
  - Should include some assertions where appropriate
	4/5%	We follow the principle of “high cohesion and low coupling”. We split the whole system into 20+ classes(and interfaces) and each plays one or two specific roles. We have dumped Server class and let each class to directly associate with another, for example we make Customer be able to directly create BikeRentalOrder objects (if they successfully checked out) This reduces longer range cou

		pling as compared with having Server gene rating BikeRentalOrder for Customer. We have used assertions in many methods. For example, setCurrentOrder(BikeRentalOr der) for Bike, we assert this.CurrentOrder = = null because one bike can only serve one customer at one time frame. Other methods implementing assertions ar e getType() , RemoveDatesFrom_BookingD ates(...) in Bike, overlaps(...) in DateRange an d etc.

- Readability
  - Code should be read able and follow codin g standards
  - Should supply javado c comments for **Loca tion** and **DateRange** classes
	4/5%	Use explicit naming for variable and provid e sufficient comment to clarify ambiguous c odes.
		
		We provide Javadoc comments for Location and DateRange

Q4. Report	**10%**	

- Revision to design
  - Design document cla ss diagram matches i mplemented system
  - Discuss revisions ma de to design implem entation stage
	5/5%	Design document class diagram matches i mplemented system
		
		We discussed most of the changes made t o uml class diagram. Some changes (such a s typo) are too trivial to discuss.

- Self-assessment - Attempt a reflective s elf-assessment linked to the assessment cri teria	4/5%	We have reflected all our works linked to th e assessment criteria.