CS3343 Software Engineering Practice

2022/23 Semester A

Group Project for Group 13

XBOX

Box Storage Management System

Analysis and Design Report

Conducted by:

DONG Jiajie	56641314
LI Xiaoyang	56638660
SHA Xinchen	56641824
ZHANG Tiantian	56645190
ZHENG Shangkun	56642570
ZHOU Yu	56642568

Dec 4, 2023

Table of content

1. Introduction

It is busy at the beginning of every vacation in the center of Student Resident since plenty of students wants to go back to hometown and left some package to CSSAUG storing service. There always a mess when students intend to rent the boxes, store items, and fetch back the package, since CSSAUG always use excel and humanity to manage the service. In fact, this inefficient management waste many times and effect.

Thus, we expected to develop a new program managing this storing service more efficiently. User can rent boxes, store items, and fetch back packages by simply moving the finger on the phone. Applying our application, only one person in charge is needed in every service point, which greatly optimize the management, save the time, money, and effect.

2. Use Case Diagram and Specification

2.1 Client-User Use Case

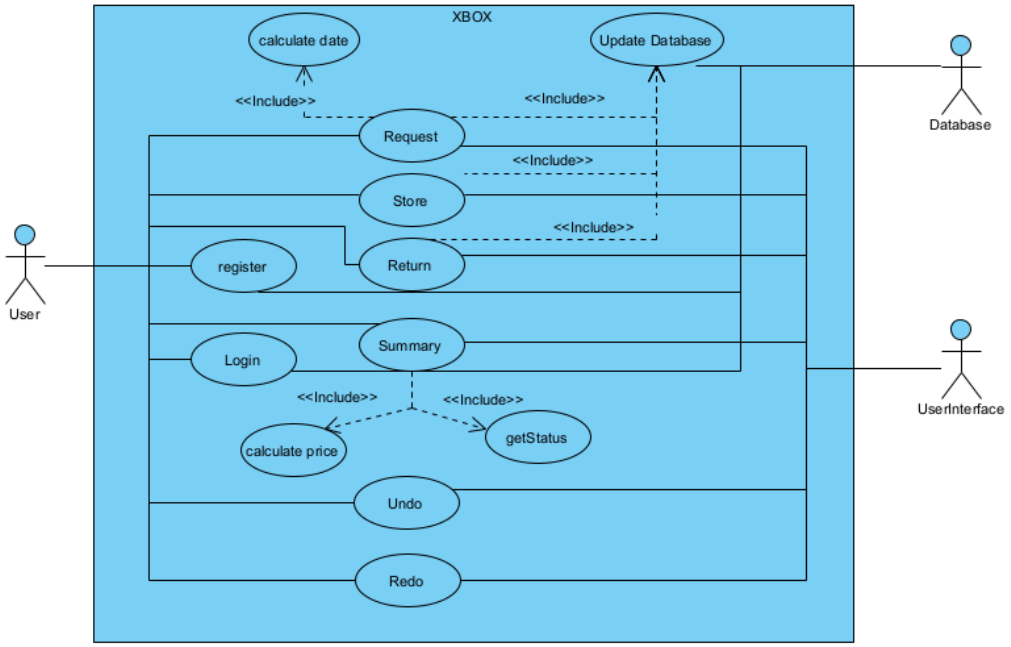

Figure 2.1 Client-User user case diagram

The Client-User Interface provides lots of convenient function for user to use. They first need to register and login into their account. They can request empty boxes to store their things on the Request page. And input the boxids to store the boxes after finishing packing. And return the empty boxes after they get back their stuff. During the whole process, they can press the summary button to check the situation of the boxes.

2.1.1 Use Case Specification – Register

Use Case Name	Register	
Actor(s)	User	
Description	This use case describes the process of a User register for an account	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user selects the function. Step 2: The user input the email, phone number and the password.	Step 3: The system creates a new account and update to database.
Alternate Course	NIL	
Precondition	The user has no account before, and the system function are normal.	
Postcondition	The User have a new account.	

2.1.2 Use Case Specification – Request

<table>
<tr><td>Use Case Name</td><td colspan="2">Request</td></tr>
<tr><td>Actor(s)</td><td colspan="2">User</td></tr>
<tr><td>Description</td><td colspan="2">This use case describe the process of a User request for a box</td></tr>
<tr><td rowspan="2">Typical course of events</td><td>Actor action</td><td>System Response</td></tr>
<tr><td>Step 1: This use case is initiated when the user select the function.

Step 2: The user input the type of rentable he want.</td><td>Step 3: The system allocate the rentable to user, calculate the due date and update to database.</td></tr>
<tr><td>Alternate Course</td><td colspan="2">Step 3a: If the system check that no wanted rentable left, the system will alert a warning.</td></tr>
<tr><td>Precondition</td><td colspan="2">The user login to system and the system function are normal.</td></tr>
<tr><td>Postcondition</td><td colspan="2">The User get the wanted rentable.</td></tr>
</table>

2.1.3 Use Case Specification - Store

Use Case Name	Store	
Actor(s)	User	
Description	This use case describe the process of a User Store his item	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user select the function. Step 2: The user input the time of fetching back items Step 4: The user pay the bill	Step 3: The system calculate the due date and return back the bill. Step5: The system check whether the bill is paid. Step 6: The system check whether the rentables with items received. Step 7: The system update to the database.
Alternate Course	Step 5a: If the bill is not paid, the system will alert. Step 6a: If the rentables are not received, the system will alert.	
Precondition	The user have requested and the system function are normal.	
Postcondition	The User store the rentables with items in storage .	

2.1.4 Use Case Specification - Return

<table>
<tr><td>Use Case Name</td><td colspan="2">Return</td></tr>
<tr><td>Actor(s)</td><td colspan="2">User</td></tr>
<tr><td>Description</td><td colspan="2">This use case describe the process of a User fetch back the items</td></tr>
<tr><td rowspan="2">Typical course of events</td><td>Actor action</td><td>System Response</td></tr>
<tr><td>Step 1: This use case is initiated when the user select the function.

Step 2: The user rise the request for the rentables with items.

Step 4: The user get the items and return the rentables.</td><td>Step 3: The system find the rentables in database and give to user.

Step 5: The system check whether the rentables are collected.

Step 6: The system update to database.</td></tr>
<tr><td>Alternate Course</td><td colspan="2">Step 5a: If the rentables are not collected, the system will alert a warning.</td></tr>
<tr><td>Precondition</td><td colspan="2">The user have stored some rentables and items and the system function are normal.</td></tr>
<tr><td>Postcondition</td><td colspan="2">The User get the get back the items.</td></tr>
</table>

2.1.5 Use Case Specification - Summary

Use Case Name	Summary	
Actor(s)	User	
Description	This use case describe the process of a User ask for summary	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user selects the function.	Step 2: The system searches in database. Step 3: The system returns the summary and print the information to the console
Alternate Course	NIL	
Precondition	The user login to system and the system function are normal.	
Postcondition	The User get the summary.	

2.2 Admin-User Use Case

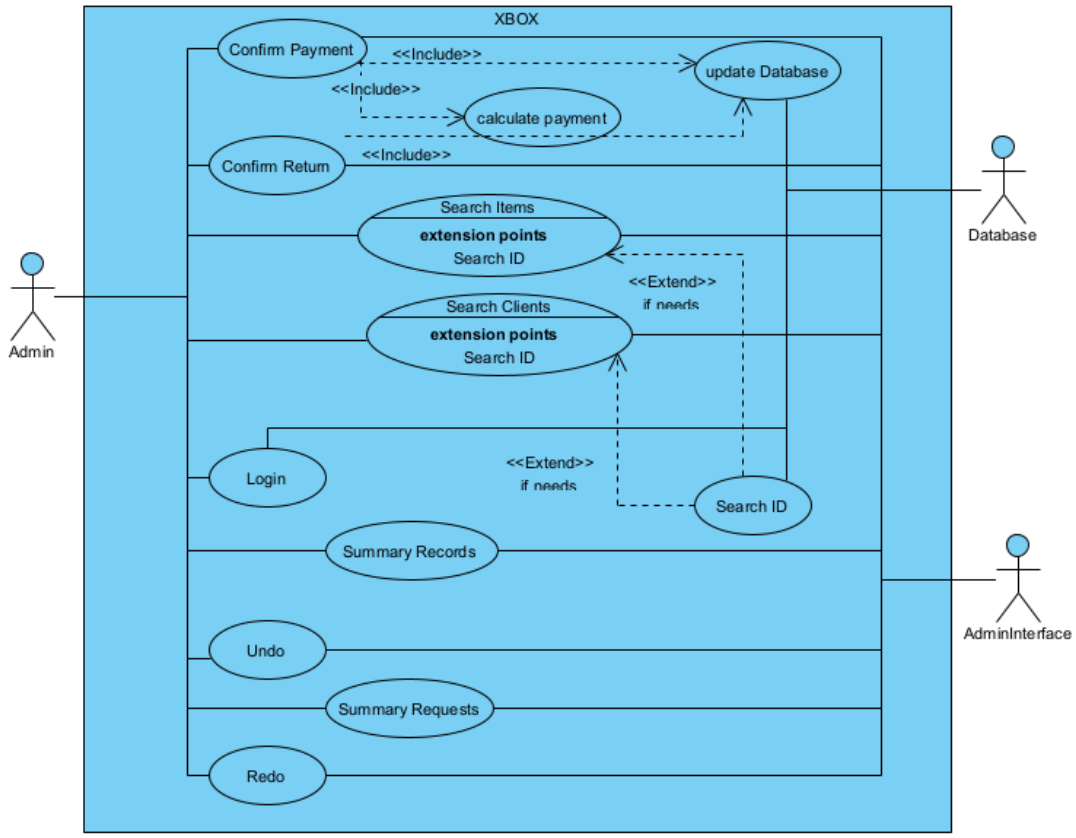

Figure 2.2 Admin-User use case diagram

The Adin-User Interface provides lots of convenient function for admin to use. They need to login into the admin account which already exists in the Database. The Admin can input the client id to know the amount of the bills and check whether they pay the correct amount. And input the client id and boxid to get the empty boxes returned by the users. The System also provides details search functions.

2.2.1 Use Case Specification – Login

Use Case Name	Login	
Actor(s)	User/Admin	
Description	This use case describe the process of a User login to system	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user/Admin select the function. Step 2: The user input his email and password	Step 3: The system check the correctness of the information
Alternate Course	Step 3a: If the system check that the password is incorrect, the system alerts a warning.	
Precondition	The user/Admin have an account in database and the system function are normal.	
Postcondition	The User/Admin login to system.	

2.2.2 Use Case Specification – Confirm Payment

Use Case Name	Confirm Payment	
Actor(s)	Admin	
Description	This use case describe the process of a Admin to confirm payment	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the Admin select the function. Step 4: The admin click the button.	Step 2:The system calculate the amount for payment. Step 3: The system return the result and rise the confirm button
Alternate Course	Step 2a: If the system check that the bill haven't been paid, the system rise an alert.	
Precondition	The user login to system and the system function are normal.	
Postcondition	The Admin confirm the payment	

2.2.3 Use Case Specification – Confirm Return

Use Case Name	Confirm Return	
Actor(s)	Admin	
Description	This use case describes the process of a admin to confirm return	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the admin selects the function. Step 3: The admin clicks the confirm button	Step 2: The system returns back the result and rise the confirm button.
Alternate Course	Step 2a: If the system check that the rentable haven't been returned, the system rise an alert.	
Precondition	The Admin login to system and the system function are normal.	
Postcondition	The Admin confirm whether the rentable have been returned.	

2.2.4 Use Case Specification – Search Item

Use Case Name	Search Item	
Actor(s)	Admin	
Description	This use case describe the process of a admin to search items	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the admin select the function.	Step 2: The system return the corresponding searching result.
Alternate Course	Step 1a: If needed, the admin enter the ID of the item.	
Precondition	The admin login to system and the system function are normal.	
Postcondition	The Admin get the wanted items' information	

2.2.5 Use Case Specification – Search Clients

Use Case Name	Search Clients	
Actor(s)	Admin	
Description	This use case describe the process of a admin to search clients	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the admin select the function.	Step 2: The system return the corresponding searching result.
Alternate Course	Step 1a: If needed, the admin enter the ID of the client.	
Precondition	The admin login to system and the system function are normal.	
Postcondition	The Admin get the wanted clients' information	

2.2.6 Use Case Specification – Summary Records

Use Case Name	Summary Records	
Actor(s)	Admin	
Description	This use case describe the process of a admin to summary record	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the admin select the function.	Step 2: The system the corresponding summary.
Alternate Course	NIL	
Precondition	The admin login to system and the system function are normal.	
Postcondition	The Admin get the summary of record	

2.2.7 Use Case Specification – Summary Requests

Use Case Name	Summary Requests	
Actor(s)	Admin	
Description	This use case describe the process of a admin to summary requests	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the admin select the function.	Step 2: The system the corresponding summary.
Alternate Course	NIL	
Precondition	The admin login to system and the system function are normal.	
Postcondition	The Admin get the summary of requests.	

3. Class Diagram

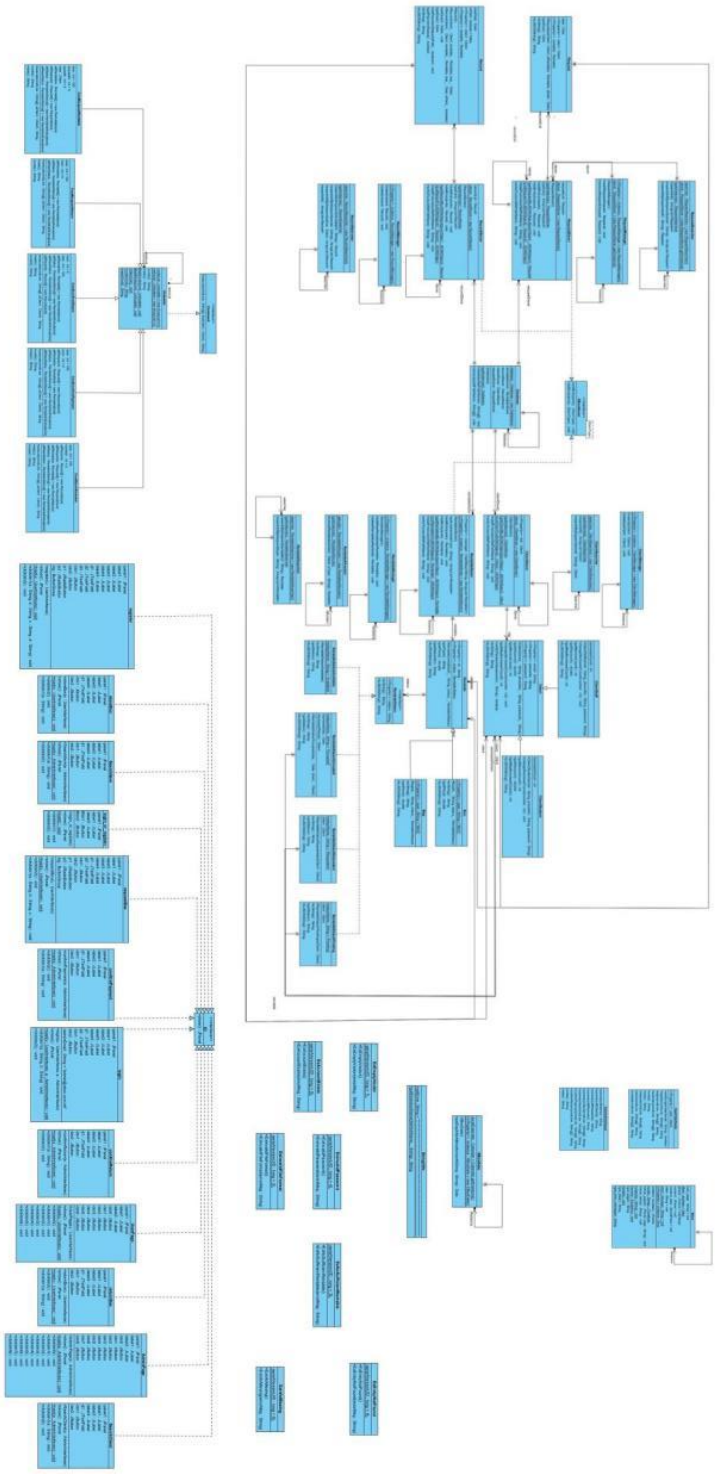

Figure 3 Class Diagram

4. Design principle

The SOLID design principles are adopted over the whole project, including the idea of single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle and dependency inversion principle. Concepts and usages of design principles mentioned above together with concrete examples will be explained in the following sections.

4.1 Open-closed principle

The open-closed principle (OCP) states that software entities should be open for extension but closed for modification. We fully appreciate the idea of allowing behaviors of an entity to be extended without its source code being modified. The idea of the OCP is clearly embodied in the scenario below.

We choose to add a abstract class called rentable, which defines different behaviors of the rentable with different definitions. Operations such as .toJsonString() are well encapsulated so that new types of rentables can be easily added to implementation without modifying any original source code.

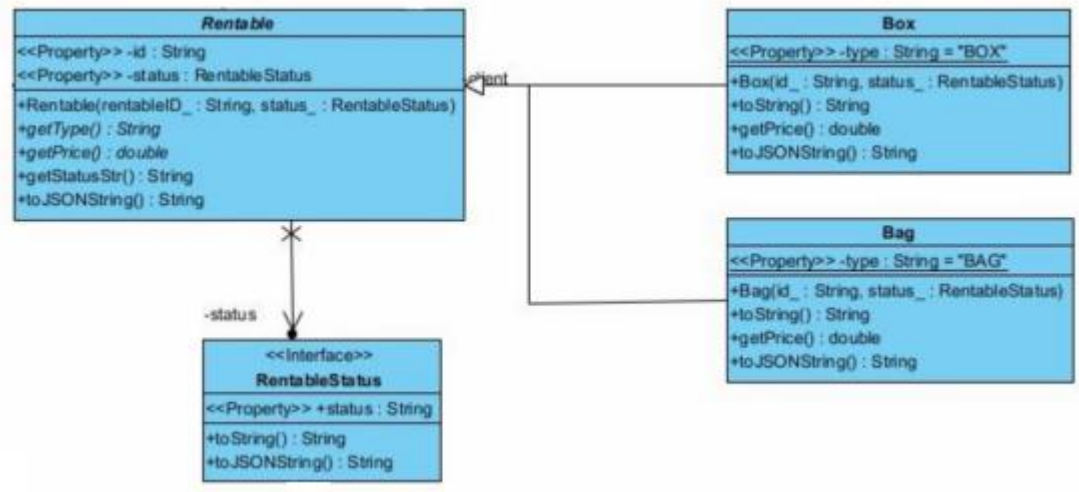

Figure 4.1 Open-closed principle

4.2 Liskov substitution principle

The Liskov substitution principle (LSP) is a particular definition of strong behavioral subtyping, stating that an instance of the super class and another instance of its subclass must be interchangeable without breaking the program. It ensures that subclasses merely extend the base class without changing its behavior. Also, subclasses are not supposed to inherit non-existing features from super classes in real interactions.

The Client abstract class is designed to define basic function of all kinds of clients, which is inherited and implemented by concrete client classes, i.e., ClientStaff, ClientStudent. Moreover, these concrete classes also define their own attributes and operations.

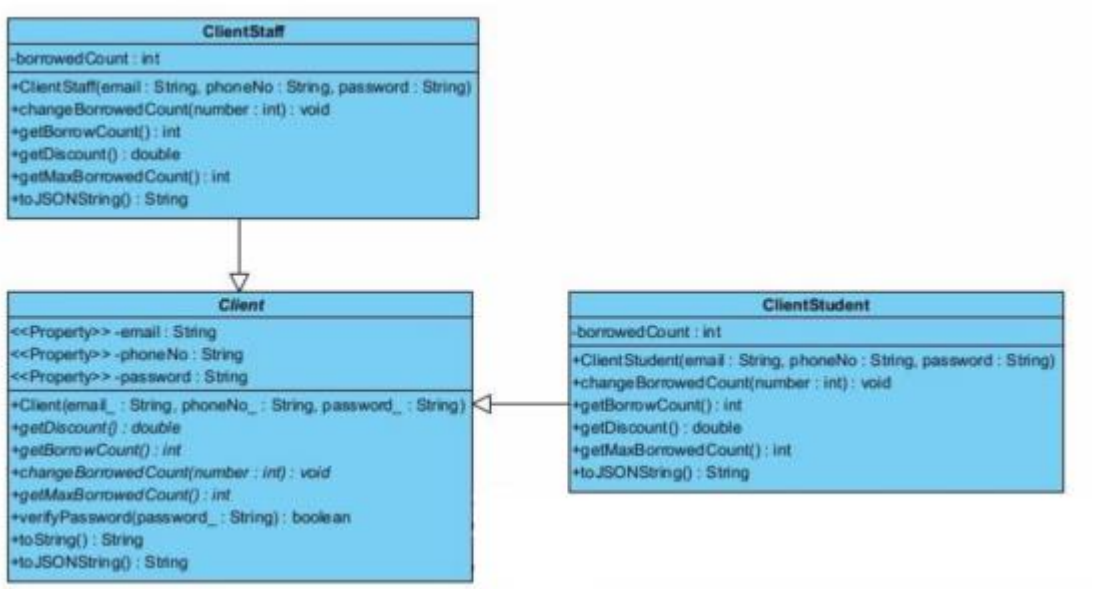

Figure 4.2 Liskov substitution principle

4.3 Single Responsibility principle

The idea behind the single responsibility principle (SRP) is that every class, module, or function in a program should have one responsibility/purpose in a program. As a commonly used definition, "every class should have only one reason to change". In fact, we divide the server for Record into three parts: searcher, manager, storer, each of which has its own responsibility of searching, managing, or storing record.

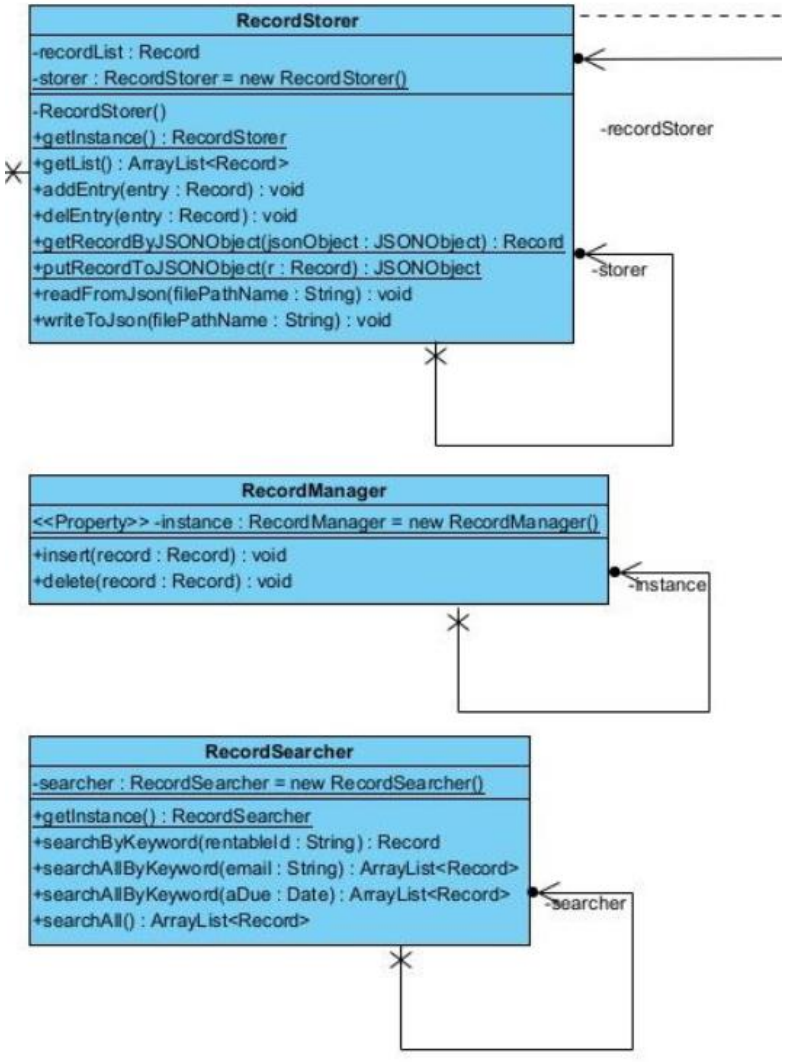

Figure 4.3 Single Responsibility principle

4.4 Dependency inversion principle

In object-oriented design, we emphasize that:
• High-level modules should not import anything from low level modules. Both should depend on abstractions.
• Abstractions should not depend on details. Concrete implementations should depend on abstractions.

In our design, the concrete Client constructor is dependent on abstract Client constructor interface. Similarly, both ClientStaff class and ClientStudent class depend on abstract Account interface.

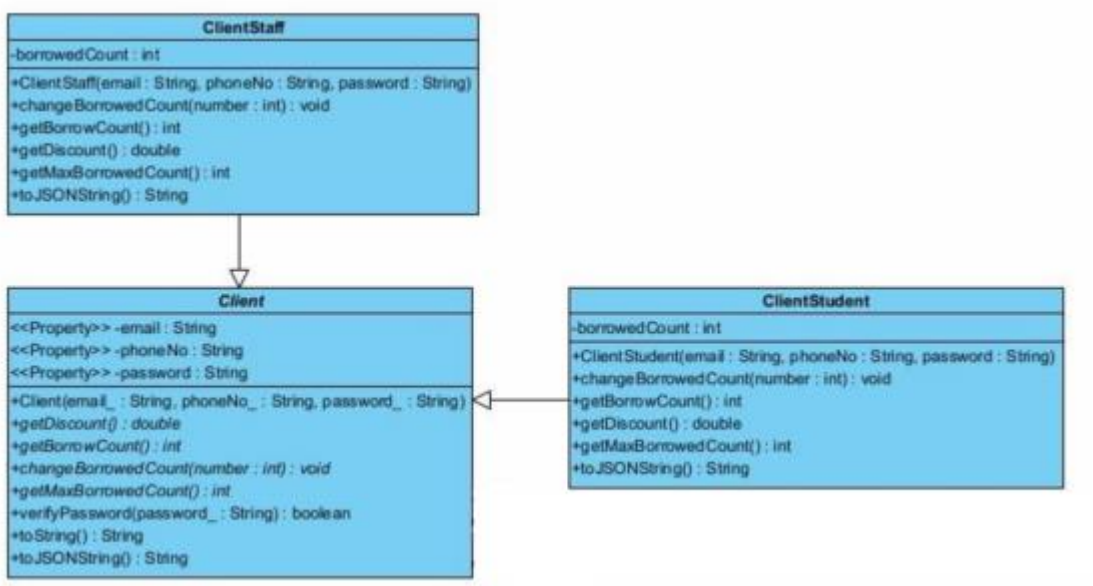

Figure 4.4 Dependency inversion principle

5. Design pattern

5.1 State pattern

The state pattern is one of the behavioural design patterns, allowing an object to alter its behaviour when its internal state changes. The object will appear to change its class. In our project's software design, we used the state pattern many times, and the figure below is a typical use case. Here is the diagram:

We encapsulate various status of rentables in RentableStatusAvailable, RentableStatusRequired, RentableStatusPending and RenatableStatusOccupied, which respectively implement the .toJsonString() and .getStatus() function defined by the uniform RentableStatus interface, and the behaviors of checking the current status of account can be delegated to one of these objects at any time.

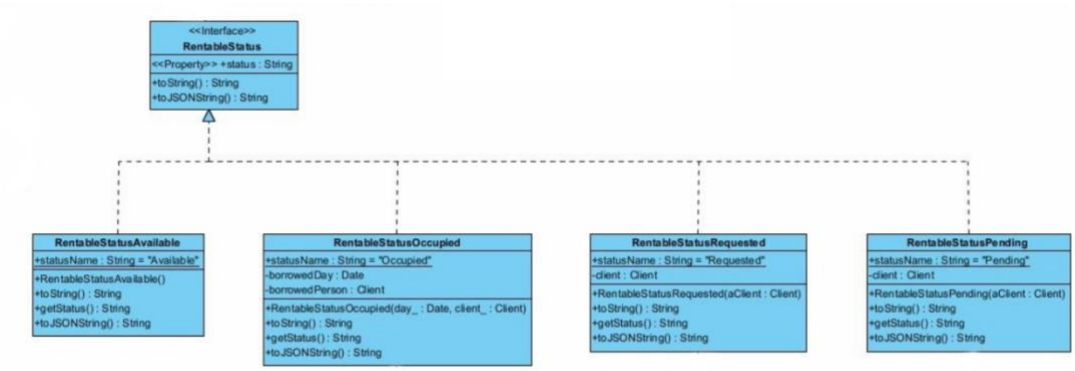

Figure 5.1 State Pattern

5.2 Command pattern

The command pattern is used to encapsulate commands (method calls) requests/commands without knowing underlying requests. It allows clients to issue different requests with ease and saving requests in a queue for reference.

We use the command pattern in our design to deal with the commands that users may send as shown below.

Moreover, we add .undo() and .redo() to the implementation of some command in need.

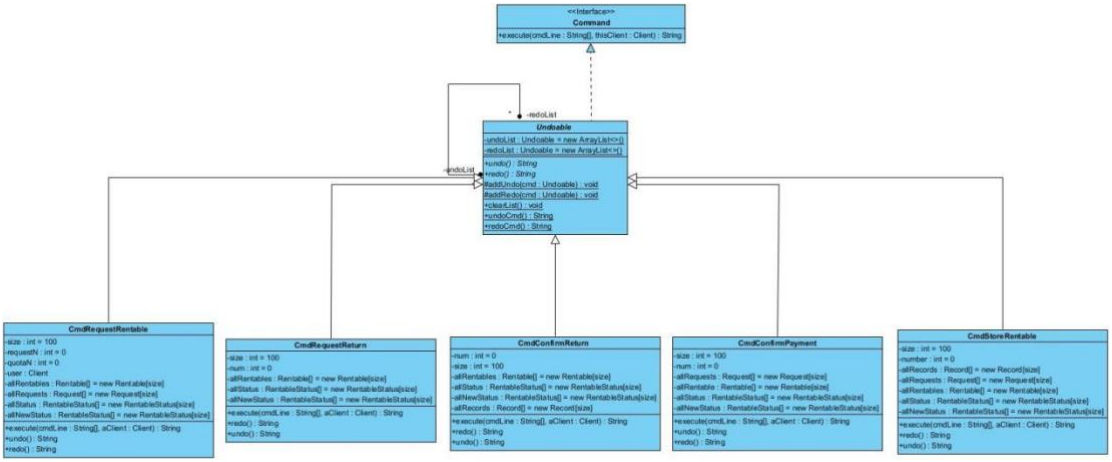

Figure 5.2 Command Pattern

5.3 Singleton pattern

The singleton pattern is used when we need to have only one instance for a class, singleton instance itself, and only one of such. We ensure that only one instance is ever created for such class to reduce the number of duplicated objects being created. We adopt singleton pattern a lot in our design such as the ClientSearcher, RentableAllocator, RecordManager.

The most typical implementation of this pattern in our system is the Database classes. Here is the diagram.

There will be only one unique database instance in this system. Any module that requires the relevant data should access this instance to get the data. This design pattern can reduce the waste of space resources and make it convenient to access the needed resources.

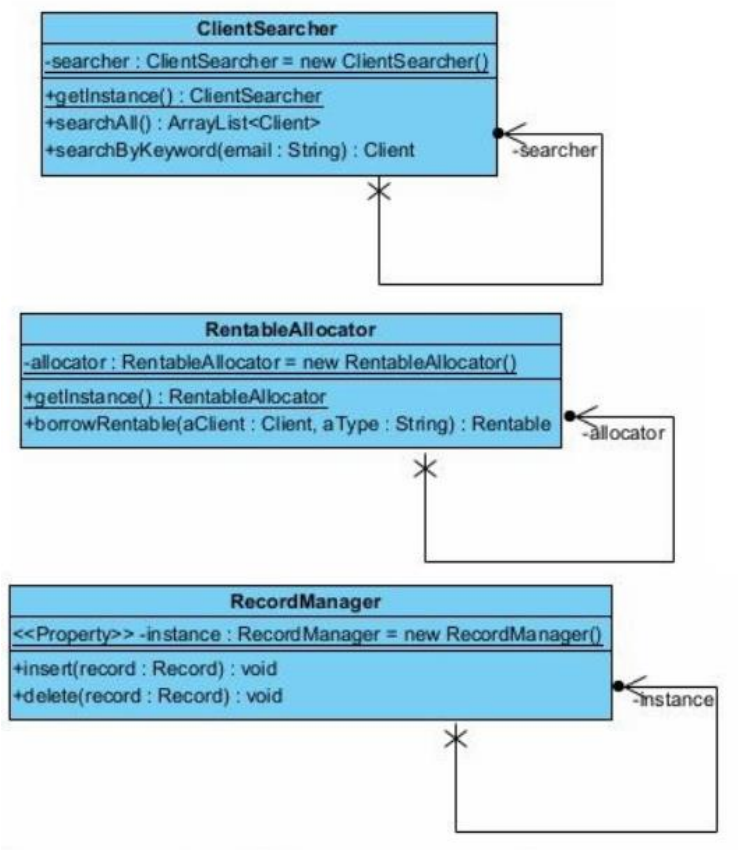

Figure 5.3 Singleton pattern

5.4 Facade pattern

The facade pattern is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.

In our design, we create AdminInterfaces for administrator to maintain the system. Also we create UserInterfaces for user to use the system. We hide the complex implementation or structure behind each method, and users can operate the system by simply calling the corresponding function in above UI.

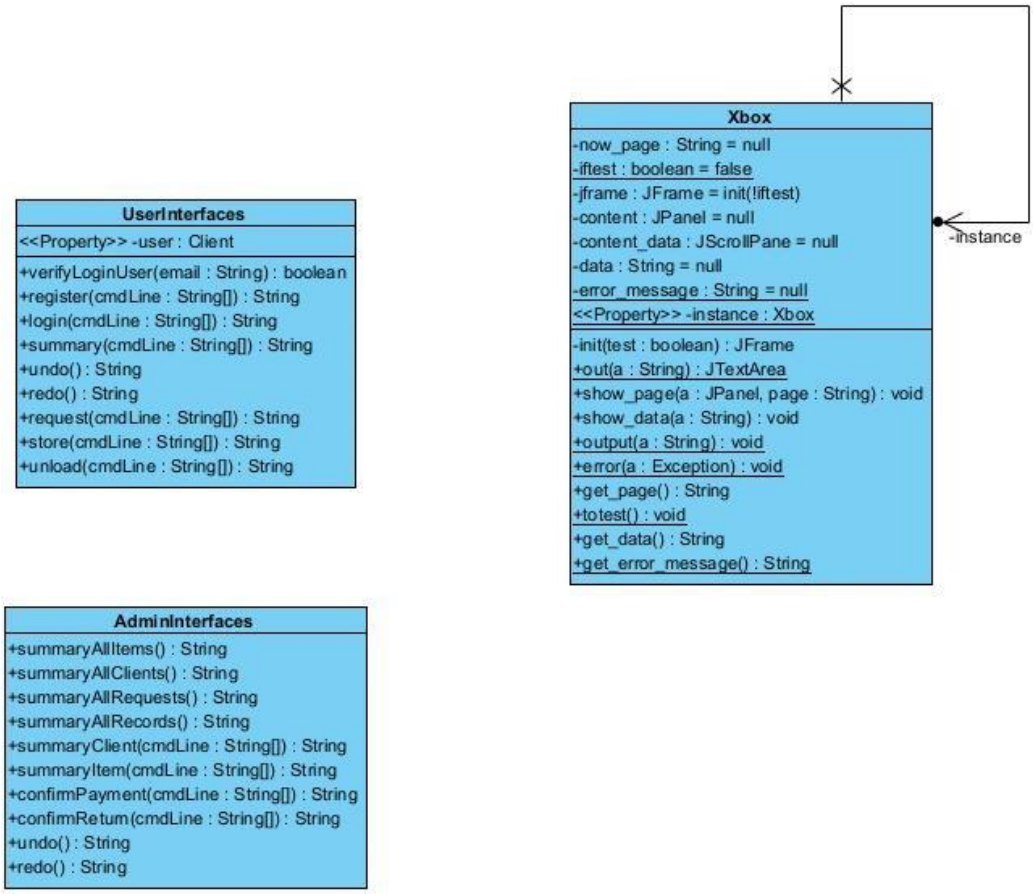

Figure 5.4 Façade pattern

6. Sequence Diagram

6.1 Account register

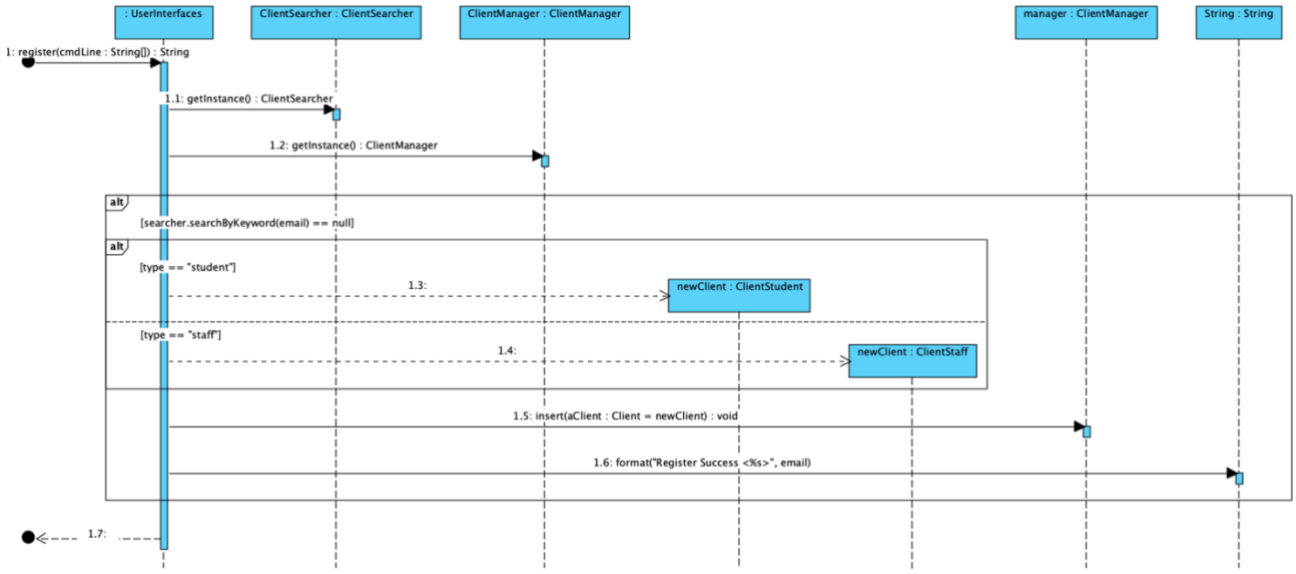

Figure 5.1 Account Register

The user's register function is mainly completed by the ClinetManager instance and the database. When the user enters his email address, desired password, their phone number and selects the type of account. The UserInterface will receive and handle the registration. It will first get the ClientSearcher to check whether there already exists an account have the same email address. After that, it will call the ClientManager instance to generate a new client according to the type of account. Then the instance will insert the client into the database. It will return a string of the result of the operation, which will be printed into the Graphical User Interface.

6.2 Login

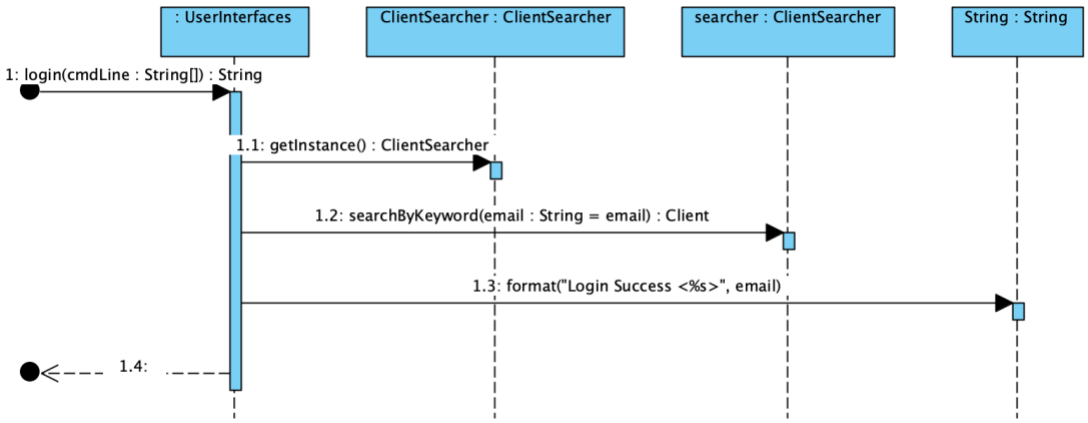

Figure 6.2 Login

The login function is mainly completed by the ClientSearcher instance. When the user types in their email address and the corresponding password. The UserInterface will call the ClientSearcher instance to search the client account from the Database and check whether the password is correct. After checking, it will return a string of the result of the operation, which will be printed into the Graphical User Interface. And the Interface will turn to the User or Admin page according to the account.

6.3 Request

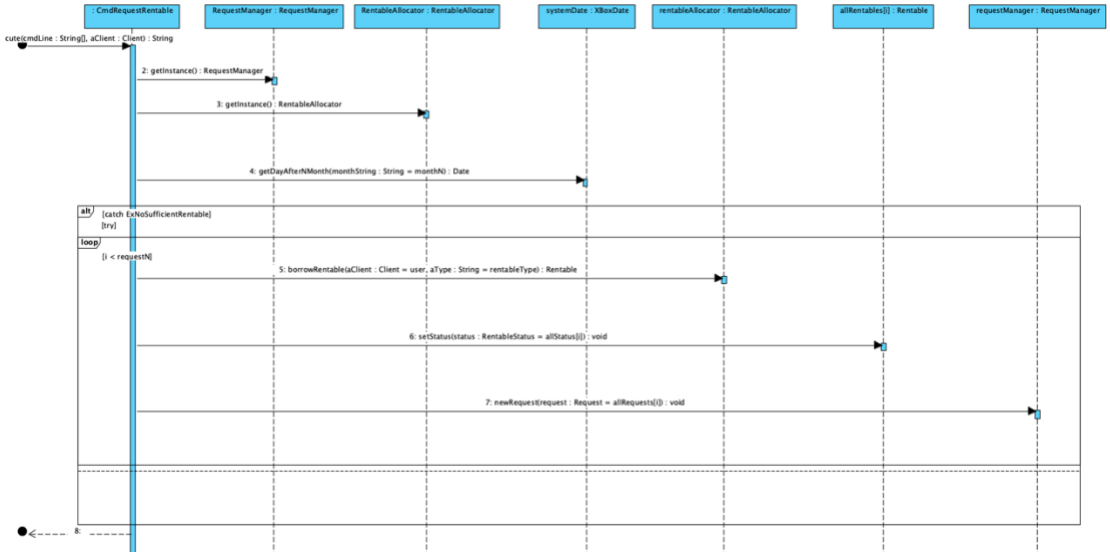

Figure 6.3 Request

The Request command is a command launched by the client user. The user types the number of boxes and the month for storage. The Interface will pass the information to the RequestManger Instance and RentableAllocator Instance. They will first check whether there exists a sufficient number of boxes. And calculate the due date for the storage. Then it will update the status of each box and update them in the Database. After that, it will record the process into the Database, which can be checked by Admin later.

6.4 Store

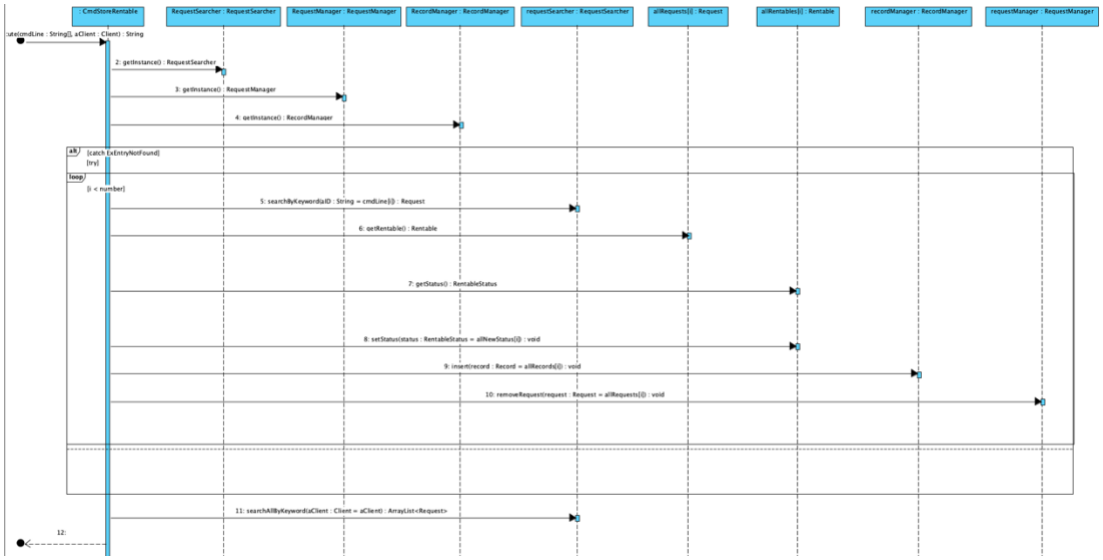

Figure 6.4 Store

The Store command is a command launched by the client user. The user types the boxids for storage. The Interface will pass the information to the RequestSearcher Instance and RentableManager Instance. They will first check the situation of the boxes, such as the current status and ownership. And calculate the total amount of the order. Then it will update the status of each box and update them in the Database. The Admin can check the pay bills of the client to check whether they pay the correct amount.

6.5 Return

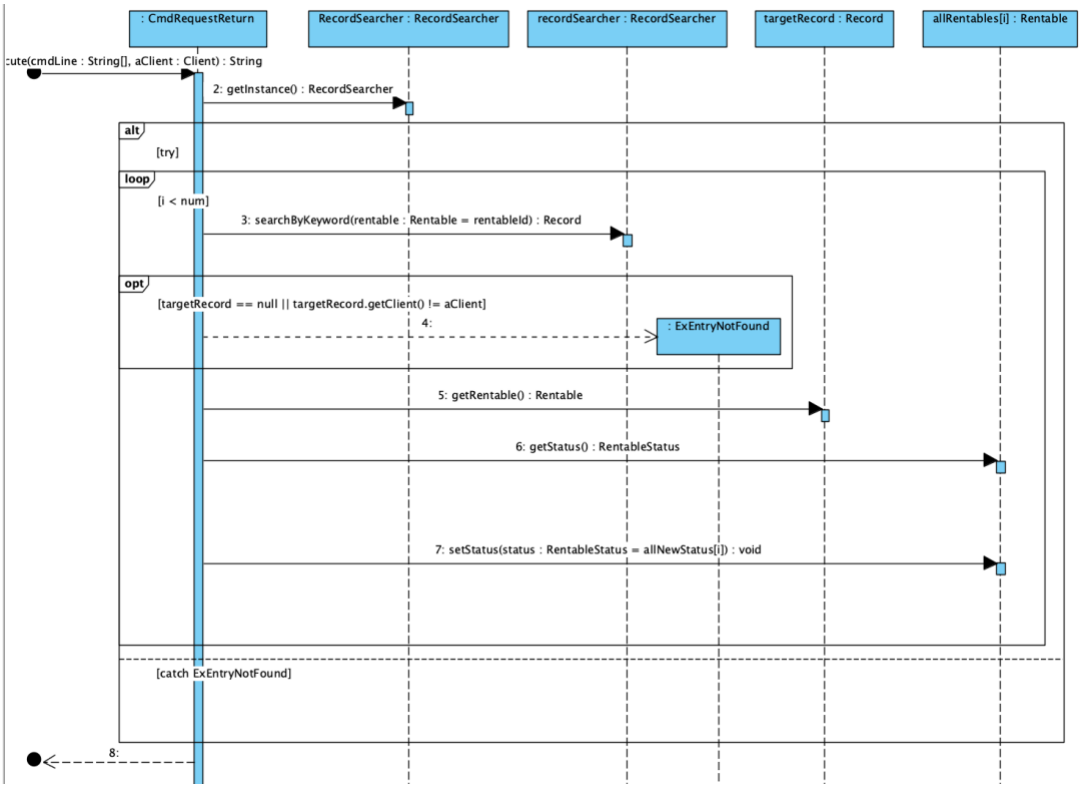

Figure 6.5 Return

The Return command is a command launched by the client user. The user types the boxids for returning empty boxes. The Interface will pass the information to the RecordSearch Instance. It will first check whether the box exists or belongs to this user. After that, it will update a targetRecord, which store in the Database, waiting for the Admin to check the actual situation. The targetRecord will disappear when the Admin confirms the correctness of the Return command.

6.6 Confirm Payment

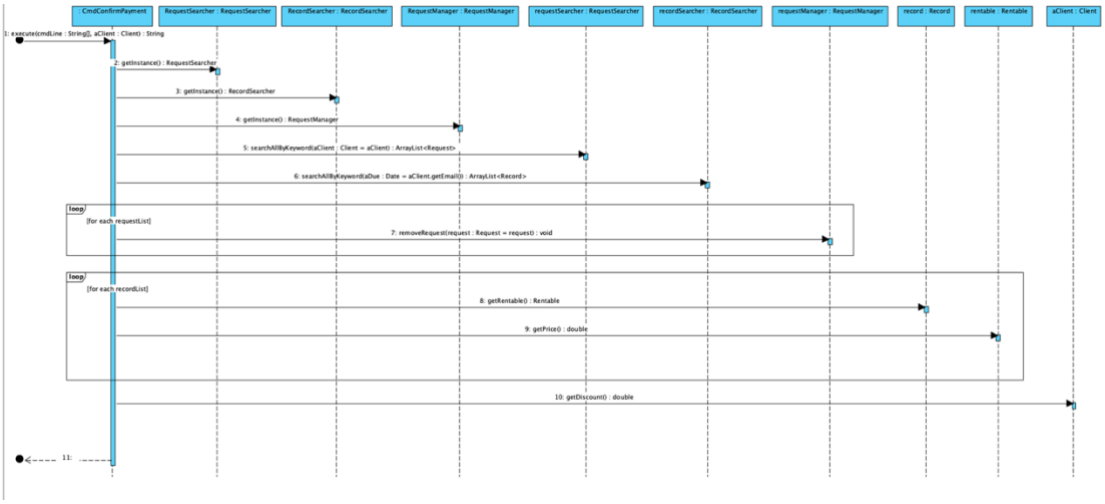

Figure 6.6 Confirm Payment

The Confirm Payment command is a command launched by the Admin user. The user types the client user id to calculate the total amount for payment. The AdminInterface will invoke the RequestSearcher and RequestManger to search the boxes belonging to the client user. And invoke RecordSearcher to call getPrice() function of each box. It will return the amount to the UI. Once the Admin confirms and checks the payment, the Interface will invoke RequestManger and RecordManger to cancel the bill and change the status in the Database.

6.7 Confirm Return

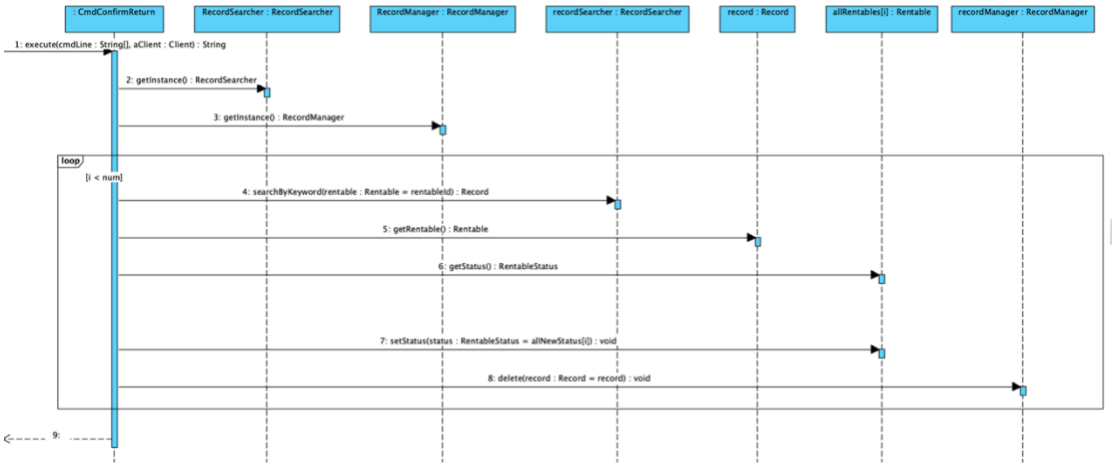

Figure 6.7 Confirm Return

The Confirm Return command is a command launched by the Admin user. The user types the client user id and boxids. The AdminInterface invokes the RecordSearcher and RecordManager to update the available status. After that, The RecordManager will delete the Return Request and update the related record in the Database.