Group Project Report

Community Service & Management App

Conducted by:

CHEN Yihuan	yihuachen6-c@my.cityu.edu.hk
GAO Nanjie	nanjiegao2-c@my.cityu.edu.hk
LIU Hengche	hengchliu2-c@my.cityu.edu.hk
WAN Zimeng	zimengwan2-c@my.cityu.edu.hk
WANG Fan	fwang247-c@my.cityu.edu.hk
WANG Ying	ywang3843-c@my.cityu.edu.hk

Department of Computer Science

香港城市大學

City University of Hong Kong

April 29, 2024

Table of Contents

1 Abstract

With the increase in the number of high-end communities, more and more communities can have service facilities dedicated to providing owners. The properties in the community can also provide owners with more and more detailed services. Nowadays, more high-end communities will be equipped with some public facilities, such as swimming pools, clubs, gyms, etc. At the same time, the community will also provide some basic maintenance and services, such as home appliance repair, exterior glass wiping, etc. In order to provide better services to owners, the property will also introduce third-party organizations to provide better community facilities and services. However, the involvement of multiple agencies will make it more difficult for owners to apply for services and make it more difficult for properties to manage owner services.

Therefore, our team designed a one-stop community service app to provide residents with more convenient services and allow the property to better manage the needs of owners. We have integrated all services that may be involved in the community into one community app. The functions involved in the app include user login, facility reservation, maintenance application, property payment, complaint submission, and visitor reservation.

In this report, we provide a comprehensive review of our community service and management app. Section 2 provides the background of our project, including project scope, process model, functional requirements, non-functional requirements, and risks and constraints. In section 3, we provide the use case diagrams and specifications. In section 4, we provide the class diagrams design. For more details, section 5 introduces sequence diagram design. Our prototype is shown in section 6. In the end, the conclusion is shown in section 7.

2 Project Introduction

2.1 Motivation of Community Service & Management App

In today's fast-paced world, where time is of the essence, fostering a strong sense of community and efficient management of community services can be a challenging task. Recognizing this need, we present our innovative one-stop Community Service and management App, designed to address these challenges and bring about positive change. With a clear set of objectives in mind, our app aims to facilitate convenience for residents, improve management efficiency, enhance community engagement, and ensure transparency in service provision and maintenance process.

2.1.1 Facilitating Convenience of Residents

One of the primary motivations behind designing our Community Service App is to provide residents with a seamless and efficient way to access various community services through a single platform. By consolidating a range of services such as maintenance requests, visitor management, and community service reservations, residents can save significant time and effort. With the app's user-friendly interface, residents can conveniently navigate through the available services, making their lives easier and more convenient.

2.1.2 Improving Management Efficiency

Streamlining the management of community services is another crucial objective of our app. We understand the challenges faced by community managers in coordinating multiple tasks, responding to resident queries, and ensuring effective service delivery. Our app automates and simplifies various administrative processes, reducing the time and effort required for managers. It provides a centralized platform for managing service requests, scheduling maintenance activities, and tracking progress, thereby enhancing overall management efficiency.

2.1.3 Enhancing Engagement of Community

Community engagement is vital for fostering a sense of belonging and unity among residents. Our app aims to promote and enhance community engagement by facilitating easy communication and interaction among residents. Through features like making complaints, residents can actively participate in community projects. This engagement not only strengthens the bonds within the community but also contributes to a vibrant and inclusive environment.

2.1.4 Ensuring Transparency

Transparency in service provision and maintenance is essential for building trust within the community. Our app addresses this objective by providing a transparent platform where residents can access information about community services, maintenance schedules, and associated payment costs. They can also track the progress of their service requests, receive notifications on updates, and provide feedback. By enabling residents to stay informed and engaged in the management process, the app fosters transparency and accountability, enhancing overall community satisfaction.

2.2 Project Scope

The Community Services Management System is a comprehensive software solution designed to streamline the reservation and management of community services and facilities. It encompasses various modules, including the user interface (UI), administrative backend, and integrated payment gateway. Residents, visitors, and community managers benefit from this system, which offers convenient features for booking services, receiving notifications, and ensuring secure transactions. The UI module provides an intuitive interface for users to explore available services, while the administrative backend empowers staff to oversee reservations and track usage. Integrated payment options enhance convenience, and stringent security measures protect user data. Overall, the system aims to solve community-related challenges efficiently and effectively.

2.3 Process Model

Our approach to developing the community service management app follows an incremental model, which allows us to add functions incrementally after establishing the basic ones. This method ensures a systematic and efficient development process, enabling us to deliver a high-quality app that meets the needs of our users.

We begin by focusing on the core functionalities of the app, like the design pattern for Facebook. These basic functions include **login and authentication**,**visitor management**, providing a secure and reliable foundation for the app. By establishing these fundamental features first, we ensure that users can access the app and their personal information and register visitors in the community with ease and peace of mind.

Once the basic functions are in place, we move on to the incremental addition of features. These functions are designed to enhance the app's capabilities and cater to the specific needs of different user roles: residents, and managers.

For residents, we gradually introduce features that facilitate convenience and engagement. This may include services such as **maintenance requests**, **community service registrations**, **make complaints**, and **make payments**. By incrementally adding these features, we ensure that residents can access and utilize each function effectively, without overwhelming them with a complex interface.

Similarly, for managers, we implement functions that streamline the management of community services. This may involve features like **service request tracking**, **maintenance scheduling**,**handle complaints**, and **handle payment**. Each function is added incrementally, allowing managers to adapt and fully utilize the capabilities of the app.

Lastly, we focus on the community as a whole, introducing features that foster transparency and collaboration. This may include **feedback and updates** that encourage interaction and participation among residents.

By following the incremental model, we can thoroughly test and refine each function before moving on to the next phase. This approach ensures that the app evolves steadily, maintaining a high level of quality while keeping the development process organized and manageable.

2.4 Functional Requirements

2.4.1 Login Module

In this system, when a new resident moves in this community, mangers will preregister a user account for new resident. In order to use the app, residents need to first use their personal information to activate their account and set their password. In case resident forget their password, they can reset their password.

2.4.2 Reservation Module

Each community can have several facilities available for resident. Due to the limited amount of facilities, resident may need to make reservation before using the facilities. In reservation module, residents can make reservation of different facilities, such as swimming pool and charging station.

2.4.3 Maintenance Module

Since communities can also provide maintenance service, such as electric devices maintenance, to resident. In maintenance system, residents can submit their maintenance request

to the system so manager and service center can receive their request and arrange service to residents.

2.4.4 Visitor Module

To guarantee the safety of community, only residents or other preregistered visitor can enter the community. Also, only people who are relative to resident can come inside. In order to enter the community, residents need to enter the information of their visitor. Manager in the back-end will check visitor information. If the information passes the manager verification, the system will generate visitor QR Code to residents. Residents can send QR Code to visitor. Before visitors enter the community, manager will scan their QR Code to verify their identity.

2.4.5 Complaint Module

In this system, resident can submit their complaint letter to manager, in which residents can select complaint type and describe the complaint Managers will read all letters and make responses to each resident. The system will keep updating the complaint handling process to resident.

2.4.6 Payment Module

Since there are bills that residents need to pay. Manager will notify residents to pay for their bills. Residents can select different type of payment method to make the payment.

2.5 Non-functional Requirements

2.5.1 Usability

Our one-stop community and management service app is designed to seamlessly integrate various services, including user profile management, maintenance scheduling, and reservation systems, all connected to a centralized database. The application is engineered to handle a significant volume of transactional and user data, requiring robust storage capabilities. To address this, we have implemented a hybrid storage solution that combines our proprietary database system with third-party cloud services. This integration necessitates the development of efficient data handling and retrieval algorithms to ensure optimal performance. Regular backups and redundant cloud storage strategies are implemented to mitigate risks associated with data loss and ensure data availability.

2.5.2 Reliability

Our app manages sensitive user information, such as personal details, complaint logs, and payment records. Protecting the confidentiality and integrity of this data is of utmost importance. We prioritize obtaining explicit user consent before accessing any personal data and employ state-of-the-art encryption methodologies to safeguard user information. Our system is fortified with robust cybersecurity measures and dedicated professionals to defend against potential cyber threats. Regular security audits and updates are performed to maintain the security and privacy of user data.

2.5.3 System Reliability and Performance

Ensuring the reliability and performance of our app is crucial to provide a seamless user experience. The system is designed to handle concurrent interactions from a diverse user base without compromising performance. We prioritize system responsiveness and uptime, enabling users to file complaints, schedule maintenance, and make reservations without delays. Our network infrastructure is optimized to minimize latency and provide a consistent experience for users regardless of their location. Additionally, efficient search algorithms are employed to deliver quick and relevant results, ensuring a smooth and satisfactory user experience.

2.5.4 User Experience

User experience is a critical aspect of our one-stop community and management service app. We prioritize creating an intuitive and user-friendly interface that allows users to navigate through different features effortlessly. The app employs responsive design principles, enabling a seamless experience across various devices and screen sizes. We conduct extensive user testing and gather feedback to continuously improve the interface and enhance user satisfaction. Additionally, personalized settings and preferences are available to tailor the app experience to individual user needs.

2.5.5 Availability

Availability refers to the probability that a system will be available for a specified period of time. The community service and management system needs to ensure that users can access the system anytime and anywhere. A backup and redundant design is required to achieve high availability to avoid single points of failure and system downtime. In addition, a 24-hour duty system and emergency response mechanisms need to be in place, as well as multiple channels of technical support.

2.6 Risks and Constraints

2.6.1 System Integration

The community service management app integrates a multitude of services including user profile management, maintenance scheduling, and reservation systems, all interfacing with a centralized database. The application is engineered to accommodate a substantial amount of transactional and user data, requiring extensive storage capabilities. To address these needs, we have architected a hybrid storage solution that amalgamates our proprietary database system with third-party cloud services. This integration necessitates the development of advanced data handling and retrieval algorithms to ensure seamless performance. To mitigate risks associated with data loss from system failures or physical damages, regular backups and redundant cloud storage strategies are essential. The system architecture must be fortified to support these mechanisms, ensuring data integrity and availability

2.6.2 Data Governance

Our app manages sensitive user information, including personal details, complaint logs, and payment records. Ensuring the confidentiality and integrity of this data is paramount. As a preventative measure against misuse, we are committed to obtaining explicit user consent before accessing personal data and implementing state-of-the-art encryption methodologies. The utilization of data is solely for enhancing user engagement and experience. Moreover, the system will employ dedicated cybersecurity professionals to fortify the network against malicious incursions, establishing a robust defense against potential cyber threats. Regular security audits and updates will be instrumental in maintaining the sanctity of user data.

2.6.3 System Robustness

The robustness of the application is critical, as it must be capable of handling concurrent interactions from a diverse user base without compromising on performance. It is imperative that the system remains operational and responsive under heavy loads, ensuring that user activities such as filing complaints, scheduling maintenance, and making reservations are processed without delays. The responsiveness extends to users operating from different geographies, necessitating the implementation of optimized network protocols and potentially distributed database solutions to minimize latency. Additionally, the search functionality within the app must be powered by efficient algorithms to provide quick and relevant results, thereby ensuring a smooth and satisfactory user experience.

3 Use Case Diagrams and Specifications

3.1 Login Module

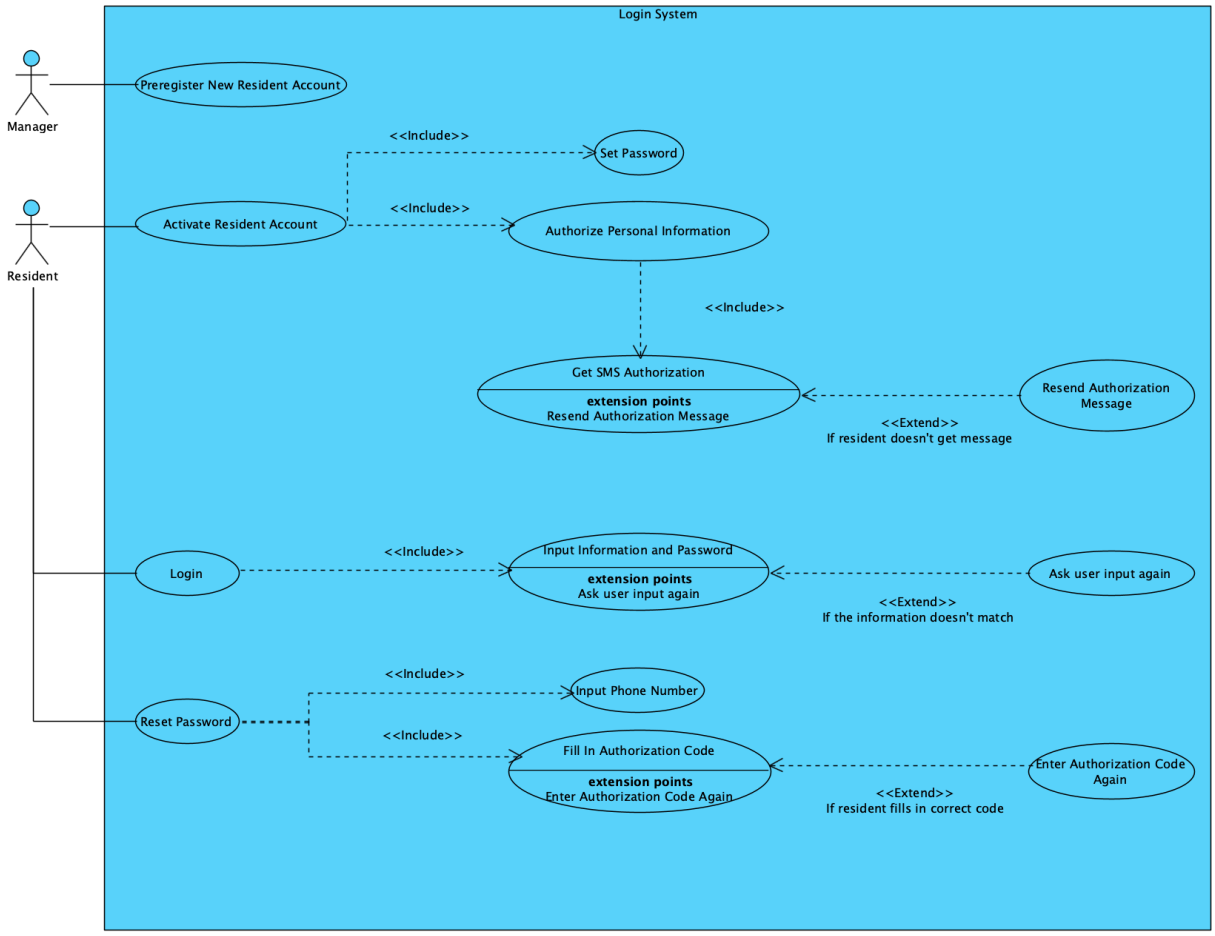

Figure 1: Login System

Firstly, Manager **Creates New Resident Account**. This functionality allows the manager to create a new account for a resident, typically involving capturing necessary information and generating unique credentials.

Once the account is created, Residence **Activates Account**. This functionality involves activating the resident account, granting access to the app's features and services. Activation may require verification of the resident's identity or confirmation from the management.

Resident can proceed to **Resident Log In**. This functionality enables the resident to securely access their account by providing the appropriate login credentials, such as a username and password. Successful login grants the resident access to their personalized account interface.

In case a resident forgets their password, the app offers the **Resident Reset Password** functionality. This feature allows the resident to initiate the password reset process, typically involving phone number verification and notification if it is the original one.

3.1.1 Use Case Specification - Preregister New Resident Account

Use Case Name:	Preregister New Resident Account	
Actor(s):	Manager	
Description:	This use case outlines the steps for a manager to preregister a new resident account.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Step 1: Resident processes to Preregister New Resident Account.	Step 2: System successfully preregisters a new resident account.
Pre-condition:	There is a new resident.	
Post-condition:	Manger creates a non-activated new account for newly arrived resident.	

Table 1: Preregister New Resident Account

3.1.2 Use Case Specification - Activate Resident Account

Use Case Name:	Activate Resident Account	
Actor(s):	Resident	
Description:	This use case describes the process by which a resident activates their preregistered account.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: The manager successfully preregister a new resident Account, and the resident initiates the account activation process.	
		Step 2: System invokes the use case "Authorize Personal Information".
		Step 3: System invokes the use case " Get SMS Authorization".
	Step 4: Resident enters the required personal information into the system	
		Step 5: System invokes the use case "Set Password".
	Step 6: Resident inputs a new password for the account.	
		Step 7: System activates the account
Alternative Courses:	Step 3a: [Extension point: if resident doesn't get message, the system will resend the authorization message again.]	
Pre-condition:	The resident has a preregistered account.	
Post-condition:	The resident account is active.	

Table 2: Activate Resident Account

3.1.3 Use Case Specification - Login

Use Case Name:	Login	
Actor(s):	Resident	
Description:	This use case describes the process by which a resident logs into their account.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident processes to login.	
		Step 2: System invokes the use case "Input Information and Password".
	Step 3: Resident enters the required personal information and password into the system.	
		Step 4: System authorizes the information and log in the user.
Alternative Courses:	Step 3a: [Extension point: if the password and personal information don't match, the system will invoke use case "Ask user input again".]	
Pre-condition:	The resident has an activated account.	
Post-condition:	The resident is logged into the system.	

Table 3: Login

3.1.4 Use Case Specification - Reset Password

Use Case Name:	Reset Password	
Actor(s):	Resident	
Description:	This use case outlines the steps for a resident to reset their forgotten password.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident processes to Reset Password.	
		Step 2: System invokes the use case "Input phone number".
		Step 3: System invokes the use case " Fill in Authorization Code".
	Step 4: Resident enters the phone number and authorization code.	
		Step 5: System reset the password.
Alternative Courses:	Step 3a: [Extension point: if the authorization code is not correct, the system will invoke use case "enter authorization code again".]	
Pre-condition:	The resident has an activated account.	
Post-condition:	The resident has a new password.	

Table 4: Reset Password

3.2 Reservation System

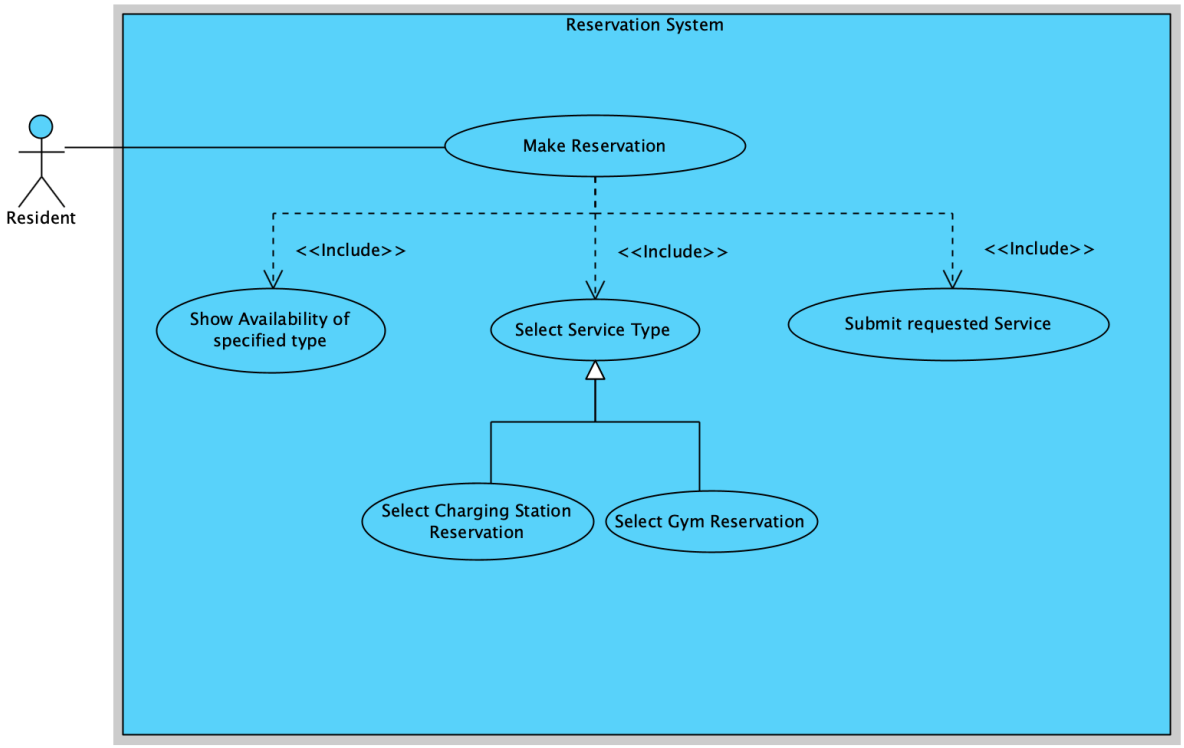

Figure 2: Reservation System

Make Reservation begins with the resident selecting the desired facility from the available options. This step allows the resident to choose the specific facility they wish to reserve, such as a charging station reservation, or a gym reservation

Once the facility is chosen, the resident proceeds to specify the reservation details. This involves providing information such as the date, time, and duration of the reservation. By specifying these details, the resident ensures that the reservation aligns with their specific requirements. After specifying the reservation details, the resident confirms the booking.

3.2.1 Use Case Specification - Make Reservation

Use Case Name:	Make Reservation	
Actor(s):	Resident	
Description:	This use case outlines the process for a resident to make a reservation for services provided by the facility, such as a gym session or a charging station.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident accesses the Reservation System and selects the "Make Reservation" option.	
		Step 2: The system prompts Resident4 to select the type of service they wish to reserve (e.g., Gym or Charging Station).
	Step 3: Resident selects the desired service type.	
		Step 4: The system displays the availability of the selected service type for various time slots.
	Step 5: Resident chooses a suitable time slot based on the displayed availability.	
		Step 6: The system prompts Resident4 to confirm the details and proceed with the reservation.
	Step 7: Resident confirms the reservation details.	
Alternative Courses:	Step 4a: [Extension point: If Resident decides to change the service type after viewing availability, the system allows Resident to go back to the service selection step.]	
Preconditions:	Resident is registered and logged into the system. The reservation system is operational and accessible.	
Postconditions:	A reservation is made for the selected service. Resident is informed of the reservation result.	

Table 5: Make Reservation

3.3 Maintenance System

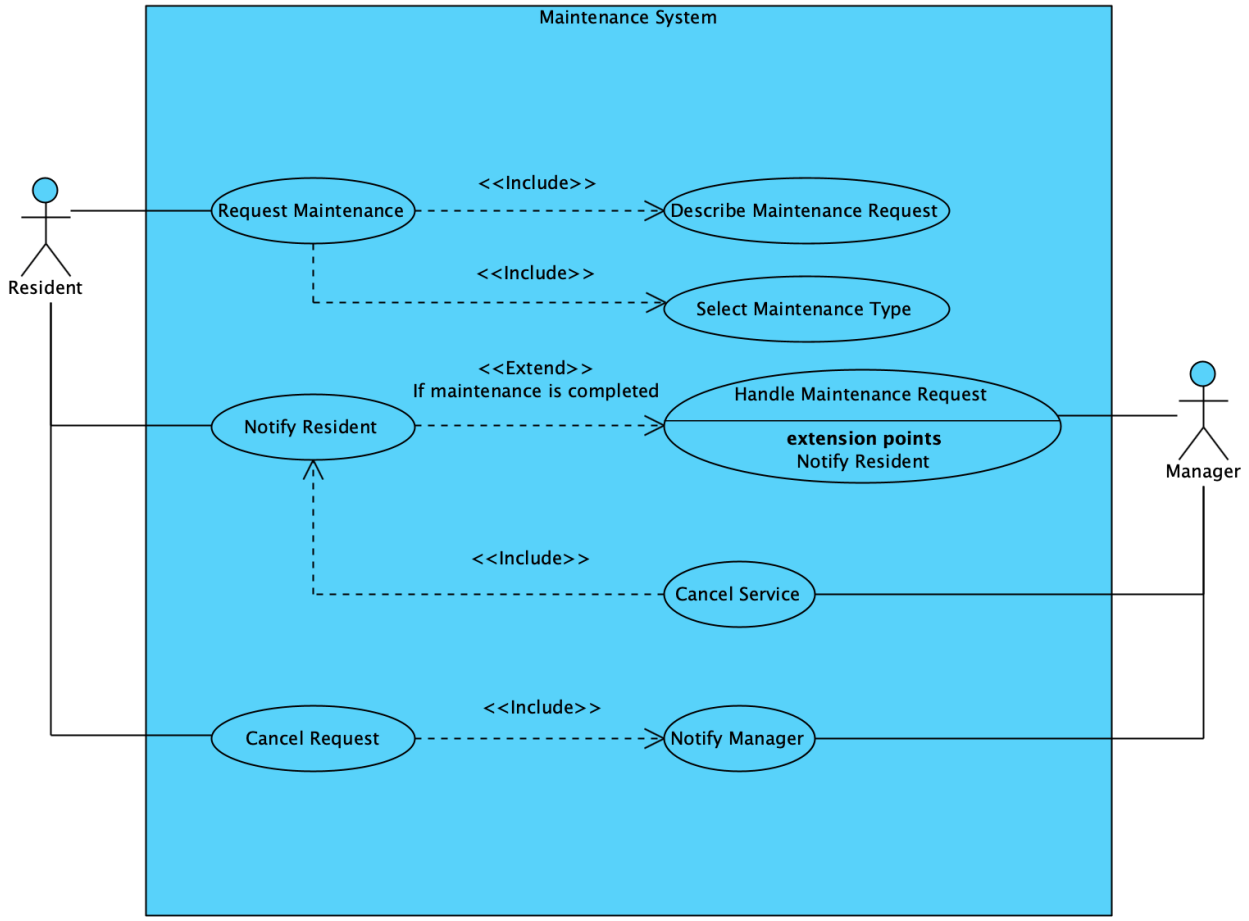

Figure 3: Maintenance System

The use case begins with the resident initiating a **Maintenance Request**, providing details and select type about the issue they are facing. This step allows the resident to report and document the maintenance concern effectively.

Upon receiving the maintenance request, the manager reviews and **Handle Maintenance Request**. This involves assessing the request, assigning resources, and updating the status to reflect the progress of the maintenance process. The manager ensures that the maintenance request is appropriately addressed in a timely manner, the manager can also **Cancel Service**.

During this process, the resident has the option to **Cancel the Maintenance Request** if the issue is resolved or no longer requires attention. This allows the resident to communicate any changes or updates regarding their maintenance needs.

3.3.1 Use Case Specification - Request Maintenance

Use Case Name:	Request Maintenance	
Actor(s):	Resident	
Description:	This use case outlines the steps for a resident to conduct a maintenance request	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident processes to Request Maintenance.	
		Step 2: System invokes the use case "Describe Maintenance Request".
	Step 3: Resident enters maintenance description.	
		Step 4: System invokes the use case "Select Maintenance Type ".
	Step 5: Resident select maintenance type.	
		Step 6: System submit the maintenance request.
Precondition:	Resident has an activated account in the system.	
Postcondition:	Maintenance request is logged in the system.	

Table 6: Request Maintenance

3.3.2 Use Case Specification - Handle Maintenance Request

Use Case Name:	Handle Maintenance Request	
Actor(s):	Manager, Resident	
Description:	This use case outlines the steps for a manager to handle maintenance request.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Manager processes to Handle Maintenance Request.	
Alternative Flows:	Step 1a: [Extension point: if maintenance is completed, the system will invoke use case " Notify Resident".]	
Precondition:	There are maintenance requests in the system.	
Postcondition:	Maintenance request is handled.	

Table 7: Handle Maintenance Request

3.3.3 Use Case Specification - Cancel Service

Use Case Name:	Cancel Service	
Actor(s):	Manager, Resident	
Description:	This use case outlines the steps for a manager to cancel a service.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Manager processes to Cancel Service.	Step 2: System invoke the use case Notify Resident.
Precondition:	There are maintenance requests in the system.	
Postcondition:	Maintenance request is canceled.	

Table 8: Cancel Service

3.3.4 Use Case Specification - Cancel Request

Use Case Name:	Cancel Request	
Actor(s):	Manager, Resident	
Description:	This use case outlines the steps for a resident to cancel a request.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident processes to Cancel Request.	Step 2: System invokes the use case Notify Manager.
Precondition:	The resident sent a request in the system	
Postcondition:	Maintenance request is canceled.	

Table 9: Cancel Request

3.4 Visitor System

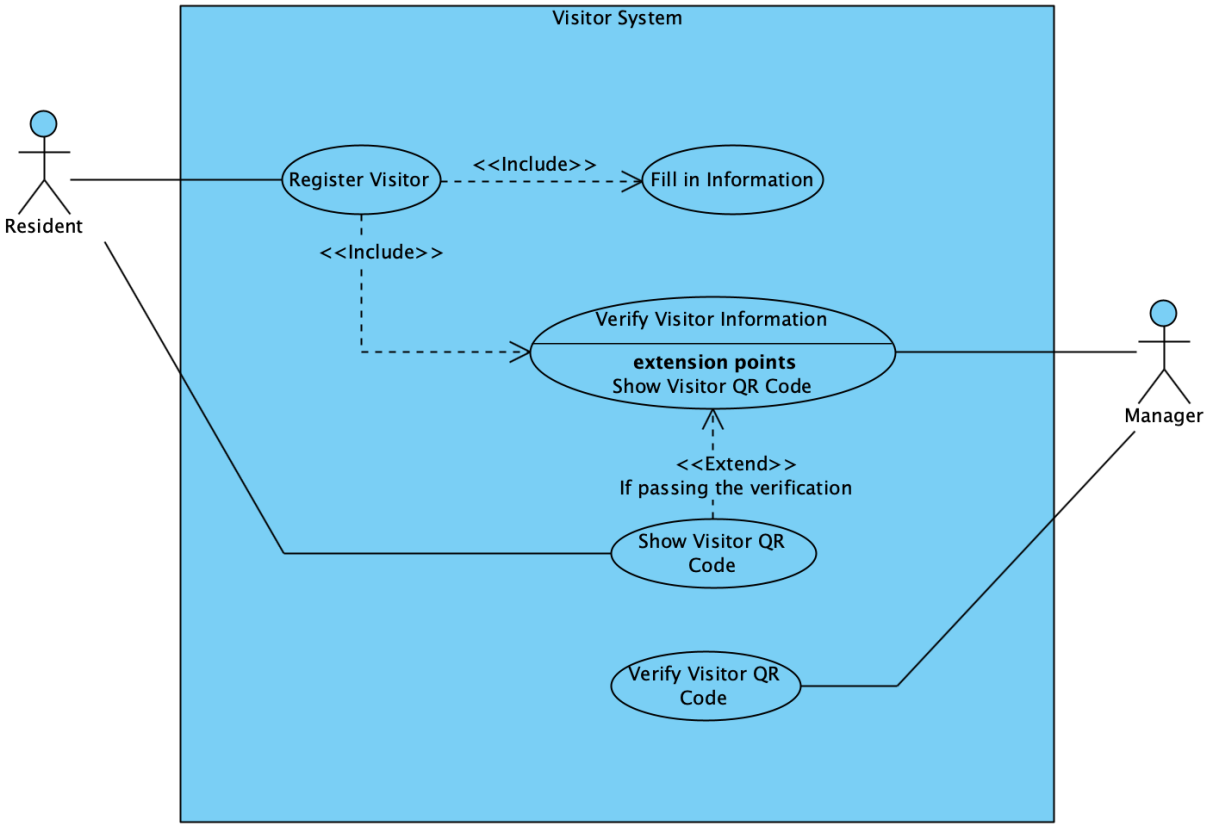

Figure 4: Visitor System

Resident **Registers Visitor**, in which resident needs to **Fill in Information**, and then Manager will **Verify Visitor Information**. If the visitor information passes the manager verification, the system will **Show Visitor QR Code** to Resident. Residents can send the screenshot of QR Code to visitors. When visitors arrive community, they will show their QR Code to Manager. Manager will **Verify Visitor QR Code**.

3.4.1 Use Case Specification - Register Visitor

Use Case Name:	Register Visitor	
Actor(s):	Resident, Manager	
Description:	This use case describes the process by which a resident can register a visitor into the visitor system, which is designed to control and document the access of visitors to a facility or residential area.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident initiates the visitor registration process.	
		Step 2: System invokes use case "Fill in Information".
	Step 3: Resident fills in information of visitor.	
		Step 4: System invokes use case "Verify Information".
	Step 5: Manager verifies visitor information.	
Alternative Courses:	Step 5a: [Extension point: if visitor information passes the verification, the system will invoke use case "Show Visitor Pass".]	
Pre-condition:	Resident wants to register a visitor.	
Post-condition:	Resident registers a visitor.	

Table 10: Register Visitor

3.4.2 Use Case Specification - Verify Visitor QR Code

Use Case Name:	Verify Visitor QR Code	
Actor(s):	Manager	
Description:	This use case outlines the steps for a manager to Verify Visitor QR Code.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Manager receive the visitor's QR Code.	Step 2: System invokes use case "Verify Visitor QR Code".
Pre-condition:	There is a visitor with QR Code.	
Post-condition:	Manger verify visitor QR Code.	

Table 11: Verify Visitor QR Code

3.5 Complaint System

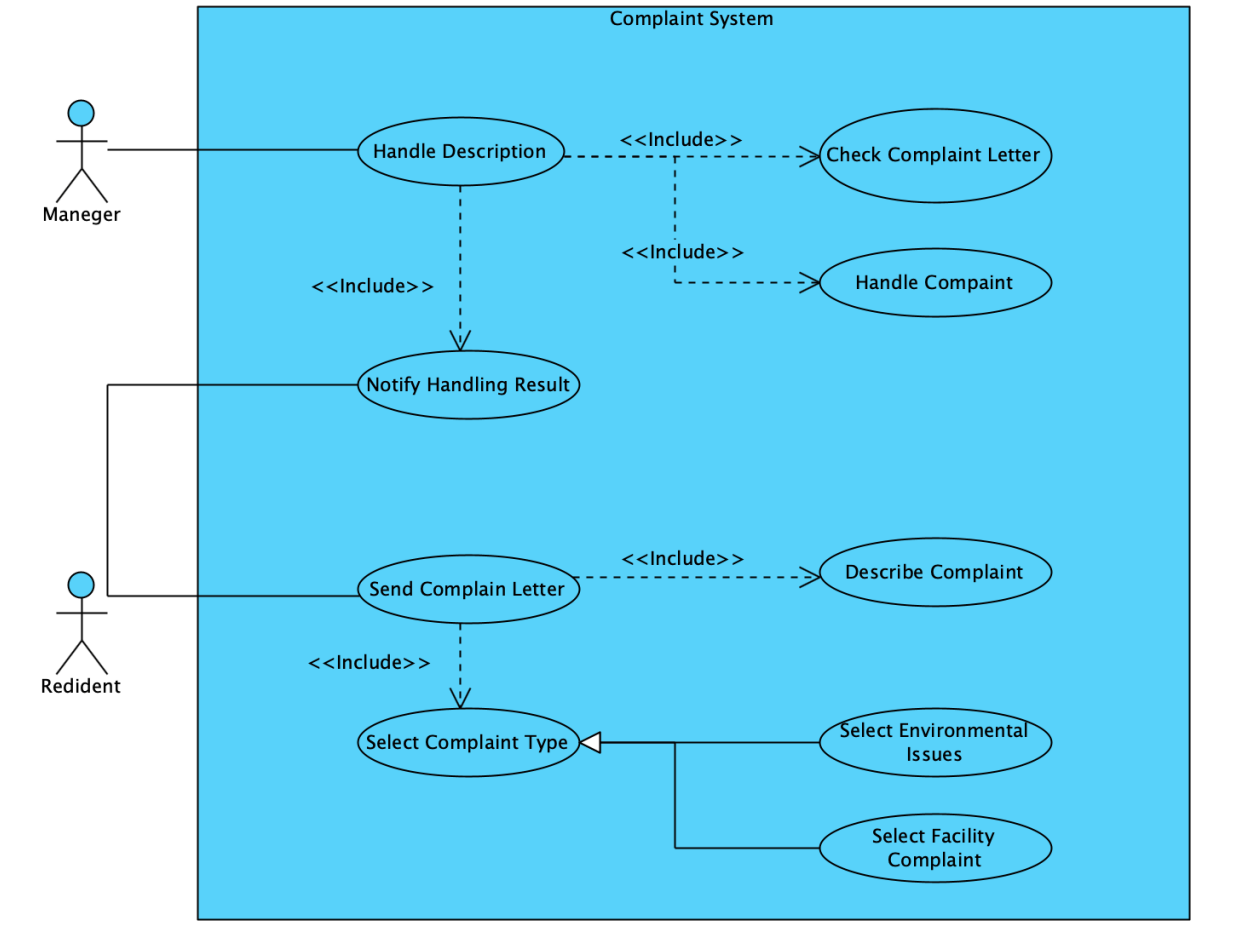

Figure 5: Complaint System

The process commences with the resident initiating a complaint by **Sending a Complaint Letter**, which may occur through various communication channels enabled by the software system. Simultaneously, the resident provides a detailed description of the issue, ensuring clarity and precision.

Upon receiving the complaint letter, the manager **Handles Descriptions** assesses its contents, meticulously reviewing the issue description and any accompanying evidence or documentation shared by the resident. Subsequently, the manager takes appropriate action to address the complaint and **Notify the Handling Result** which may include conducting investigations, coordinating with relevant parties, or implementing remedial measures.

3.5.1 Use Case Specification - Send Complaint Letter

Use Case Name:	Send Complaint Letter	
Actor(s):	Resident	
Description:	This use case describes the process by which a resident can send a complaint letter through the system.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident selects the "Send Complaint Letter" option.	
		Step 2: The system prompts the resident to select the type of complaint they are submitting ("Select Complaint Type").
	Step 3: Resident chooses the appropriate complaint category, such as "Environmental Issues" or "Facility Complaint."	
		Step 4: The system requests a detailed description of the complaint from the resident ("Describe Complaint").
	Step5: Resident provides a detailed account of the issue, including all pertinent information.	
		Step6: The system generates a confirmation message for the resident indicating successful submission.
Alternative Courses:	Step 4a: [Extension point: If the resident needs to attach additional documentation, the system provides an option to upload files.]	
Precondition:	The resident is registered and can access the Complaint System.	
Postcondition:	The complaint letter is sent and logged into the system.	

Table 12: Send Complaint Letter

3.5.2 Use Case Specification - Handle Complaint

Use Case Name:	Handle Description	
Actor(s):	Manager	
Description:	This use case outlines the process for a manager to handle the description of a complaint received from a resident.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: The manager selects the "Handle Description" option to review the details of the complaint.	
		Step 2: The system presents the complaint details, including any descriptions provided by the resident.
	Step 3: The manager assesses the information and decides on an appropriate course of action to address the complaint. Step 4: The manager may use the "Check Complaint Letter" feature if additional review of the complaint's documentation is necessary. Step 5: After reviewing, the manager proceeds to "Handle Complaint" and inputs the actions taken or to be taken into the system. Step 6: The manager notifies the handling result and gives feedback to the resident.	
Alternative Courses:	Step 3a: [Extension point: The manager provides different communication channels for the resident to provide additional information or clarification.]	
Precondition:	The complaint has been logged into the system by the resident.	
Postcondition:	The manager has reviewed the complaint's description and taken necessary action.	

Table 13: Handle Description

3.6 Payment System

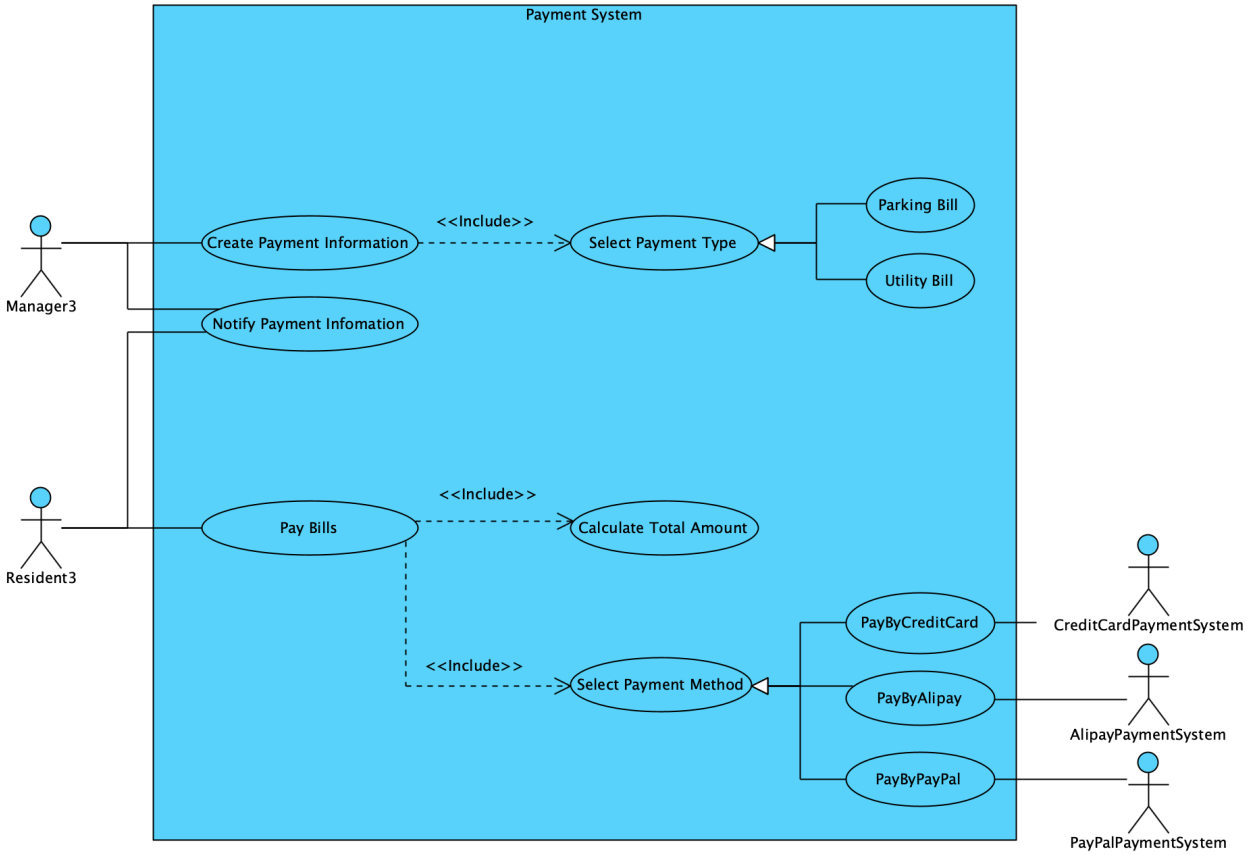

Figure 6: Payment System

The process starts with the manager **Creating Payment Information**, which encompasses tasks such as generating invoices, maintaining payment records, and monitoring outstanding balances. This step ensures accurate and up-to-date billing information.

Subsequently, the resident is prompted with the total amount to be paid, enabling transparency and allowing for a review of the charges before proceeding with the payment. This feature enhances user confidence and helps prevent any potential errors or disputes.

Finally, the resident selects a payment method from the available options, such as credit cards or PayPal, to complete the **Pay Bills**. The integration of multiple payment methods caters to the diverse preferences and needs of residents.

3.6.1 Use Case Specification - Create Payment Information

Use Case Name:	Create Payment Information	
Actor(s):	Manager	
Description:	This use case outlines the process for creating payment information for various bills in the payment system.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Step 1: Manager selects the option to create new payment information.	
		Step 2: Manager is prompted to select the type of bill for which the payment information is being created (Parking Bill or Utility Bill).
	Step 3: Manager selects the bill type	
		Step 4: The system validates and saves the payment information.
Precondition:	Manager has access rights to the payment system.	
Postcondition:	Payment information is created and stored in the system.	

Table 14: Create Payment Information

3.6.2 Use Case Specification - Notify Payment Information

Use Case Name:	Notify Payment Information	
Actor(s):	Manager, Resident, Notification System	
Description:	This use case describes how the Payment System interacts with the Notification System to inform residents about new payment information.	
Typical Course of Events:	**Actor Action**	**System Response**
		Step 1: Upon the creation of payment information by Manager3, the Payment System triggers the notification process. Step 2: The Notification System generates a notification message including the bill type, amount, and due date. Step 3: The Notification System sends the notification to the Resident.
	Step 4: Resident receives the notification and becomes aware of the new payment information.	
Precondition:	New payment information has been created in the Payment System.	
Postcondition:	Resident is notified of the payment information.	

Table 15: Notify Payment Information

3.6.3 Use Case Specification - Pay Bills

Use Case Name:	Pay Bills	
Actor(s):	Resident, CreditCardPaymentSystem, AlipayPaymentSystem, PayPalPaymentSystem	
Description:	This use case outlines the process for a resident to pay bills using the payment system.	
Typical Course of Events:	**Actor Action**	**System Response**
	Step 1: Resident selects the "Pay Bills" option from their dashboard, which displays outstanding bills.	
		Step 2:The system calculates the total amount due for the selected bill(s).
	Step 3: Resident is prompted to select a payment method (Credit Card, Alipay, PayPal).	
		Step 4: The system processes the payment through the chosen payment gateway (CreditCardPaymentSystem, AlipayPaymentSystem, or PayPalPaymentSystem).
Precondition:	The Resident has received notification of the bill. The resident has sufficient funds available for the payment.	
Postcondition:	The bill is paid.	

Table 16: Pay Bills

4 Class Diagram Design

4.1 Overall Class Diagram

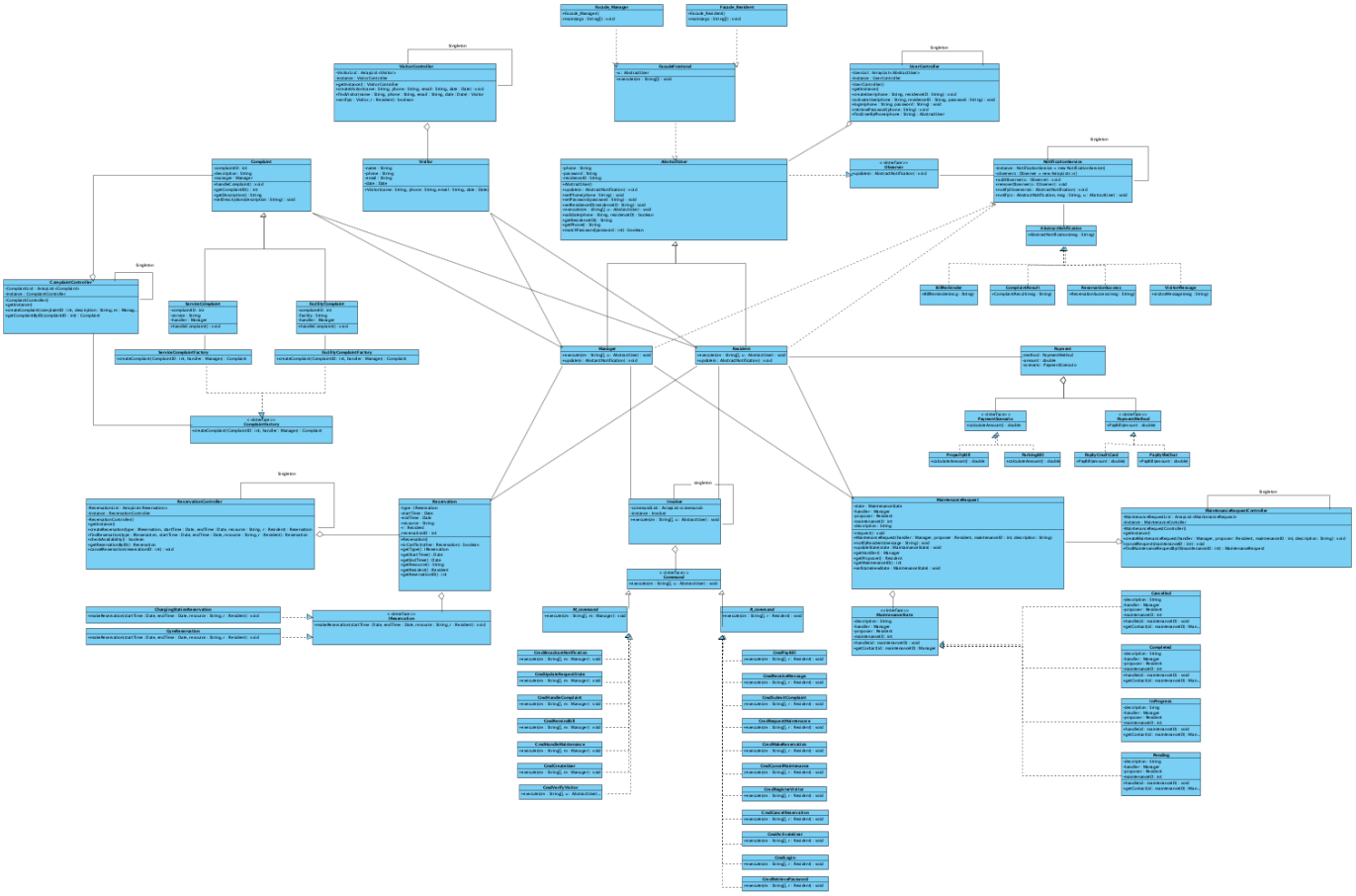

Figure 7: Overall class diagram

The figure demonstrates the overall design of our project, which will be explained in the following sections in terms of design principles and design patterns.

4.2 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.

4.2.1 Open-Closed Principle

The open-closed principle claims that the software components should be open for extension, but closed for modification, so that our code could reach high scalability but also with great security. We have implemented this idea to discourage the behavior of modification of important attributes or methods, but rather allow extension of possible future improvements. For example, the display of principle of OCP could be seen in our **payments system**

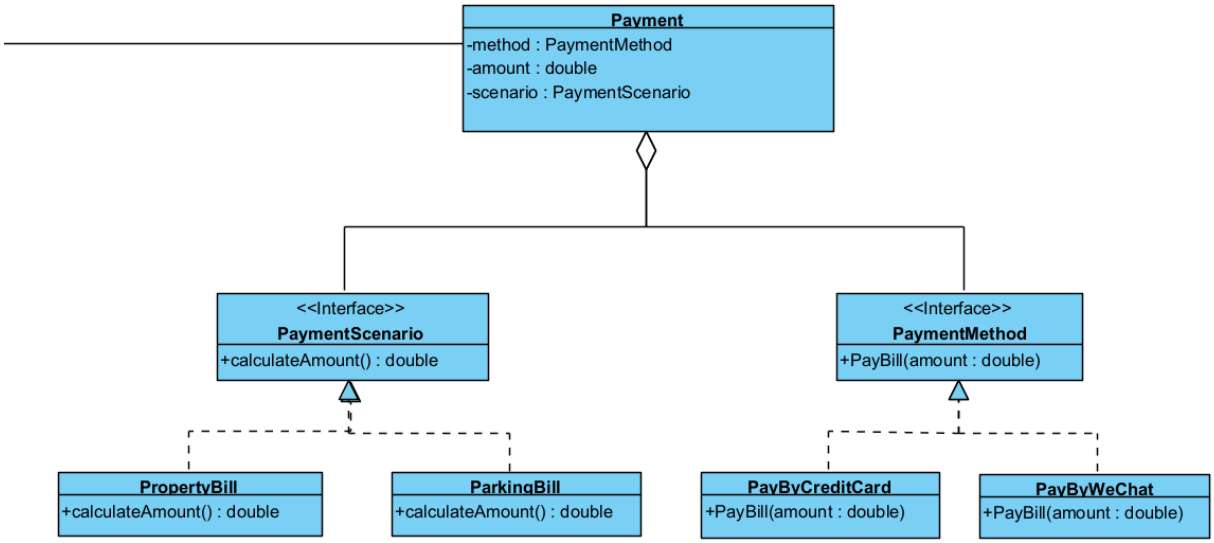

Figure 8: OCP

Instead of using lengthy and complicated if-else statements that could increase the workload if we would like to add new payment method, we implemented the OCP principle to allow users to choose their preferred **payment method** like Wechat Pay or Credit Card. Each of the concrete classes inherits the abstract class "**PaymentMethod**" and uses the **payBill()** method but they have their own specific code implementations. This makes it easy to modify one particular payment method without affecting the rest. Also, if we would like to add a new payment system, all we need to do is just create a new concrete class that inherits the abstract class.

4.2.2 Liskov Substitution Principle

LSP requires the derived subclasses must be completely substitutable for their parent class, indicating that the inheritance of classes must accurate and reasonable. This ensures that the new subclasses add to the base class's capabilities without changing its original functionality. By adhering to this principle, we can enhance our system's modules without the risk of unintended behavior during runtime.

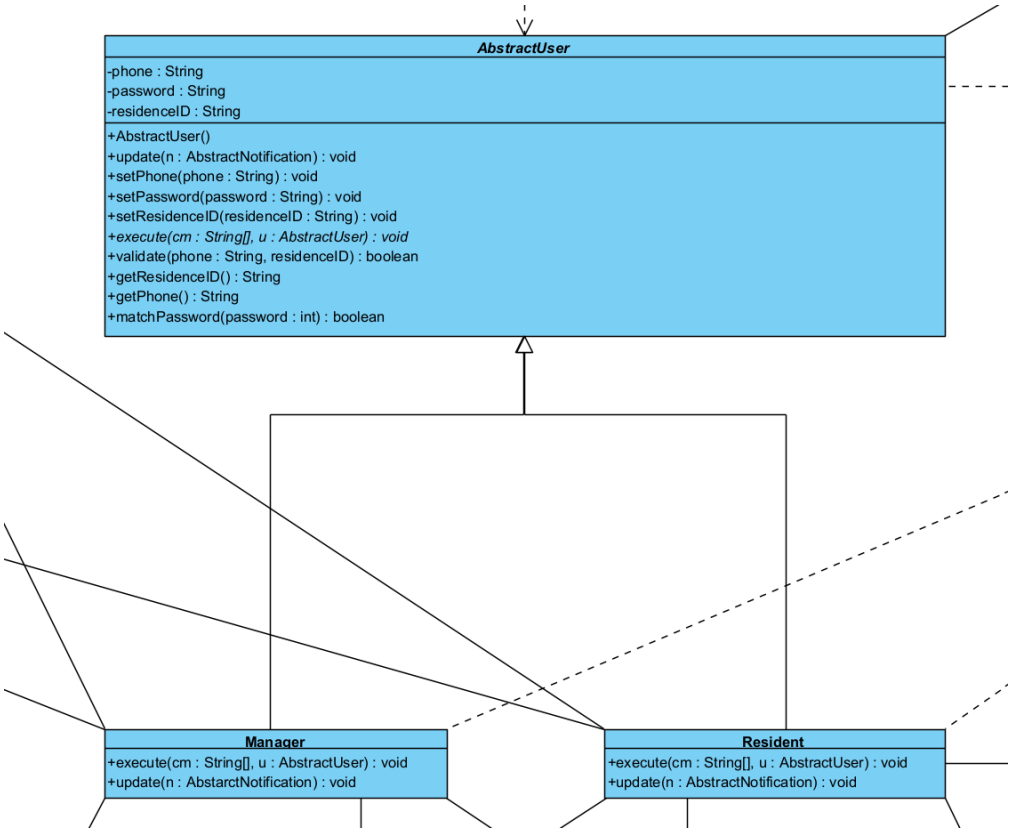

Figure 9: LSP

Our **AbstractUser** abstract superclass is created with this idea in mind. The superclass is designed to define some basic functions that all kinds of users, such as Residents and Managers, will inherit and utilize. These functions include **authentication**, **data retrieval**, and **profile management**. Subclasses like **Resident** and **Manager** can then extend these functionalities with specific features pertinent to their roles. For example, Manager might implement advanced reporting and system management capabilities, while Resident focuses on user experience and service access. This design allows us to introduce new user types, like **GuestUser** or **AdminUser**, without disrupting the existing user hierarchy or the overall system functionality.

4.2.3 Single Responsibility Principle

SRP advocates that an entity should only have one job or responsibility, which is key to achieving a modular and maintainable system, where each module addresses a specific concern. When a class is focused on a single concern, it becomes inherently easier to understand, debug and enhance without risking unintended side effects elsewhere.

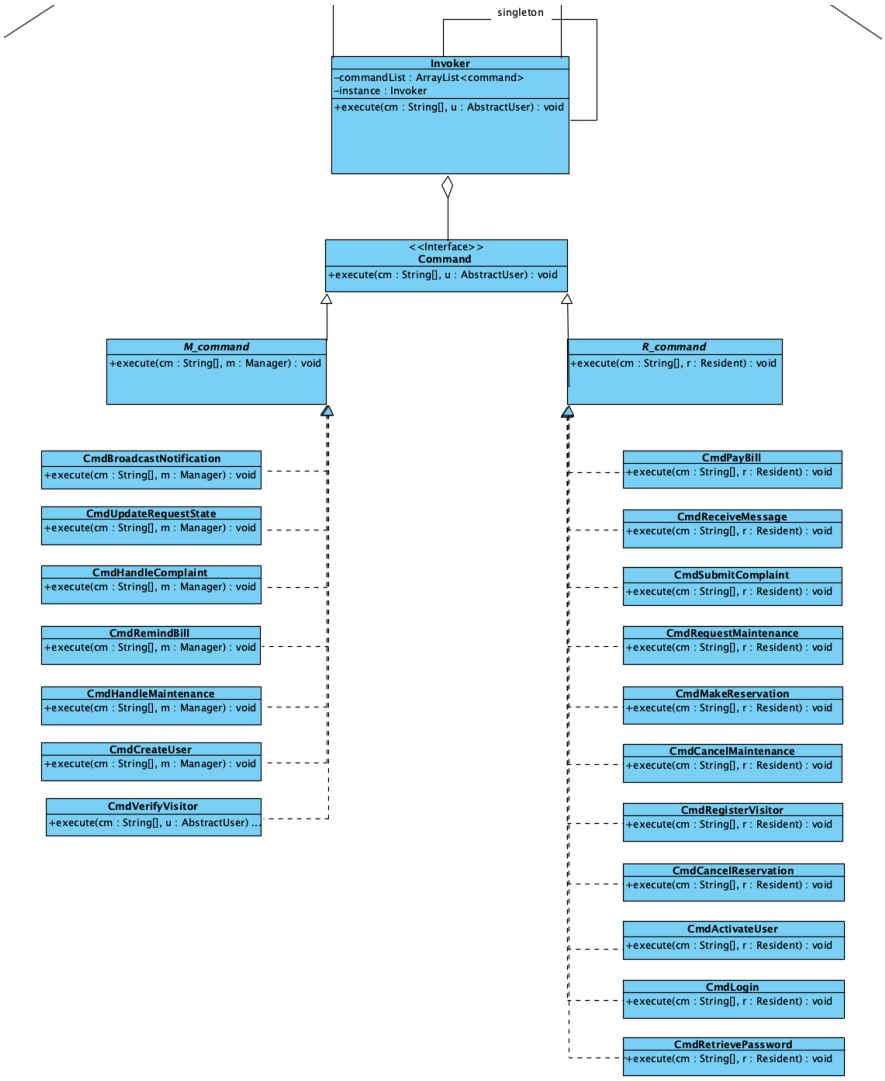

Figure 10: SRP

For example, in the command pattern, we separated the commands of Resident and those of Manager, like the **CmdRequestMaintanence()**, **CmdMakeComplaint()** for residents, **CmdUpdateRequestState()**, **CmdHandleComplaint()** for managers. By segregating the commands for Residents and Managers, each command class has a clear and distinct role, which aligns with SRP, as it allows individual command classes to be modified or extended

independently of one another, thus maintaining an organized code structure that's easy to manage and adapt over time.

4.2.4 Interface Segregation Principle

ISP supports the idea that many client-specific interfaces are better than one general-purpose interface. This principle helps to prevent "interface bloat" and ensures that implementing classes don't have to depend on methods they don't use, which can lead to cleaner, more maintainable code. By segregating interfaces according to the specific needs of clients, systems become more flexible and easier to refactor, as changes in one part of the system are less likely to impact others.

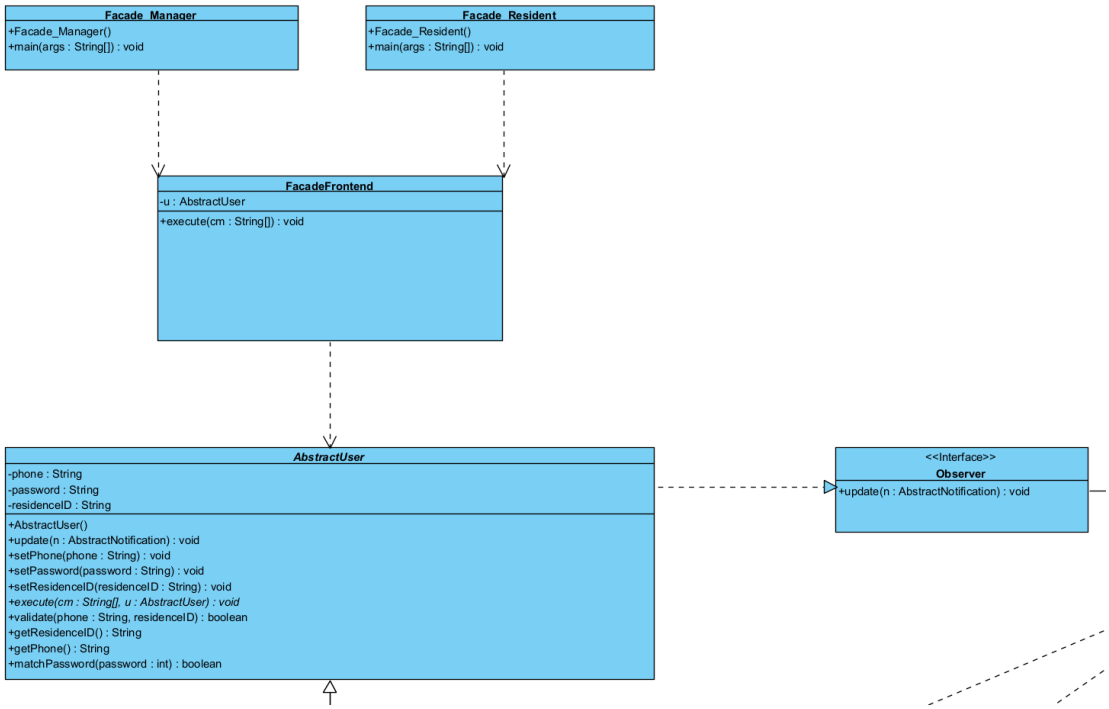

Figure 11: ISP

In our application, implementing both the facade and observer design patterns separately aligns with ISP. Instead of a monolithic interface that encompasses both patterns, creating distinct interfaces—separately for the facade pattern and for the observer pattern—ensures that classes can choose to implement only the interfaces relevant to their functionality. This separation allows for a more organized and maintainable codebase, where each pattern can evolve independently without causing ripple effects throughout the application.

4.2.5 Dependency Inversion Principle

DIP Principle states that:

- High-level modules should not depend on low-level modules.
- Abstractions should not depend on details. Details should depend on abstractions.

This principle leads to a decoupled architecture, which helps increase the flexibility and maintainability, simplify testing and reduce conflicts. By focusing on abstractions rather than concrete implementations, DIP allows for easy integration and scalability, ensuring that high-level modules remain unaffected by changes in low-level modules.

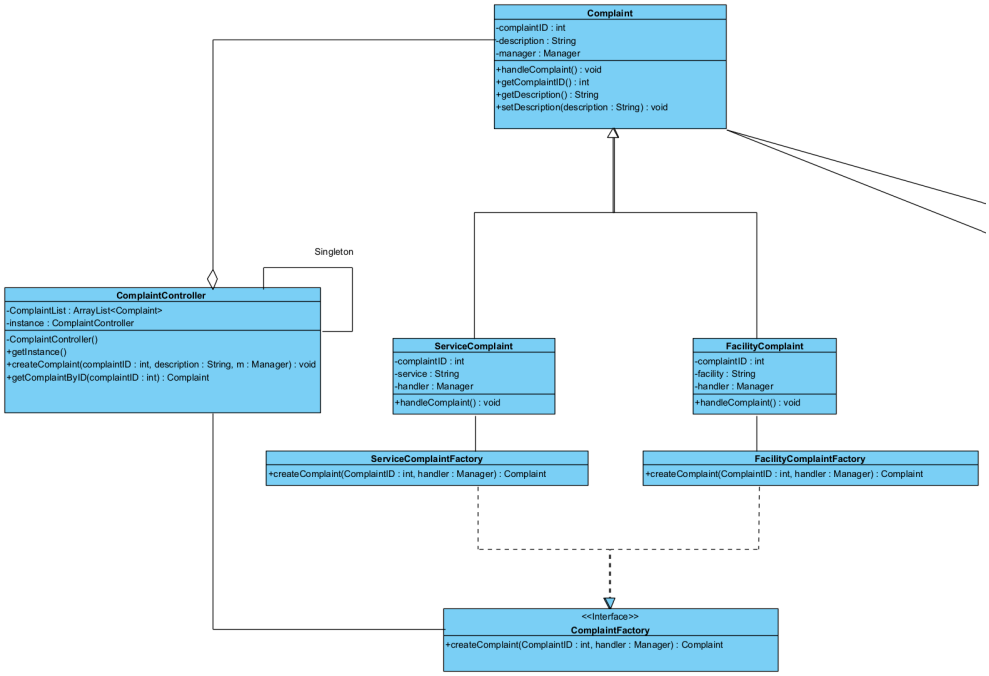

Figure 12: DIP

In the context of our **complaint** module, applying DIP means that concrete complaint constructors depend on an abstract **constructor** defined by the complaint interface. This establishes a clear hierarchy, allowing for the separation of different complaint creation processes. Such an arrangement is advantageous for future handling and modification of the **complaint module**, as it provides a structured approach to managing complaints while adhering to the principles of DIP. This structured approach ensures that any changes in the complaint handling process or the introduction of new types of complaints can be managed without significant rework, maintaining the integrity and stability of the whole system.

4.2.6 Law of Demeter

The Law of Demeter suggests that a given object should only interact with its direct components and not with internal details of other objects. This leads to a loosely coupled and more maintainable system where objects are less dependent on the internal structures of others, which reduces complexity of the system, increases the encapsulation and enhances the modularity.

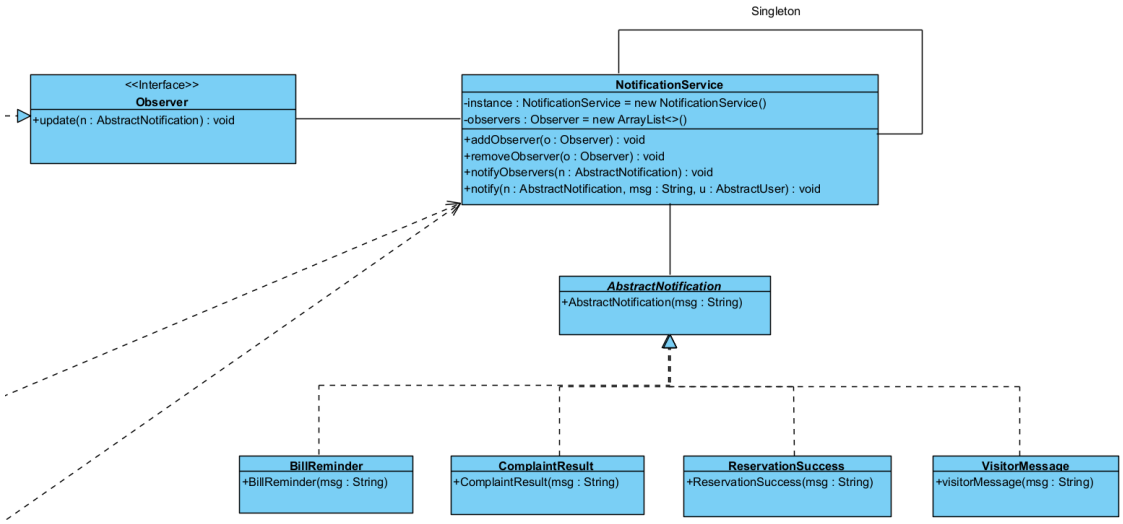

Figure 13: LoD

In our design, the observer design pattern of the **notification** module aligns with the LoD principle. In this example, the concrete classes implementing the **AbstractNotification** interface represent the objects being observed, such as visitor message and complaint result, while the Observer class defines the interface for objects that want to be notified of changes. The **NotificationService** maintains a list of observers and provides methods to add, remove and notify them. When **NotificationService** needs to notify the observers of a change, it calls **update()** method on each observer. Thus, the **NotificationService** class adheres LoD by only directly interacting with the Observer interface. It does not have knowledge of the concrete implementations of the observers and their internal details. This observer pattern aligns with the LoD by encapsulation and information hiding to outsiders, allowing objects to communicate through predefined interfaces, reducing dependencies and promoting modularity and maintainability.

4.3 Design Pattern

4.3.1 Facade Pattern

The Facade pattern is handy when there are multiple subsystems with complex interactions, and we want to expose a simple interface to the clients. By introducing a facade, the complexity of the system is hidden, making it easier for client applications to interact with the system. It is also applicable when we need to decouple a client implementation from the complex subsystem, thus promoting loose coupling and easier maintainability.

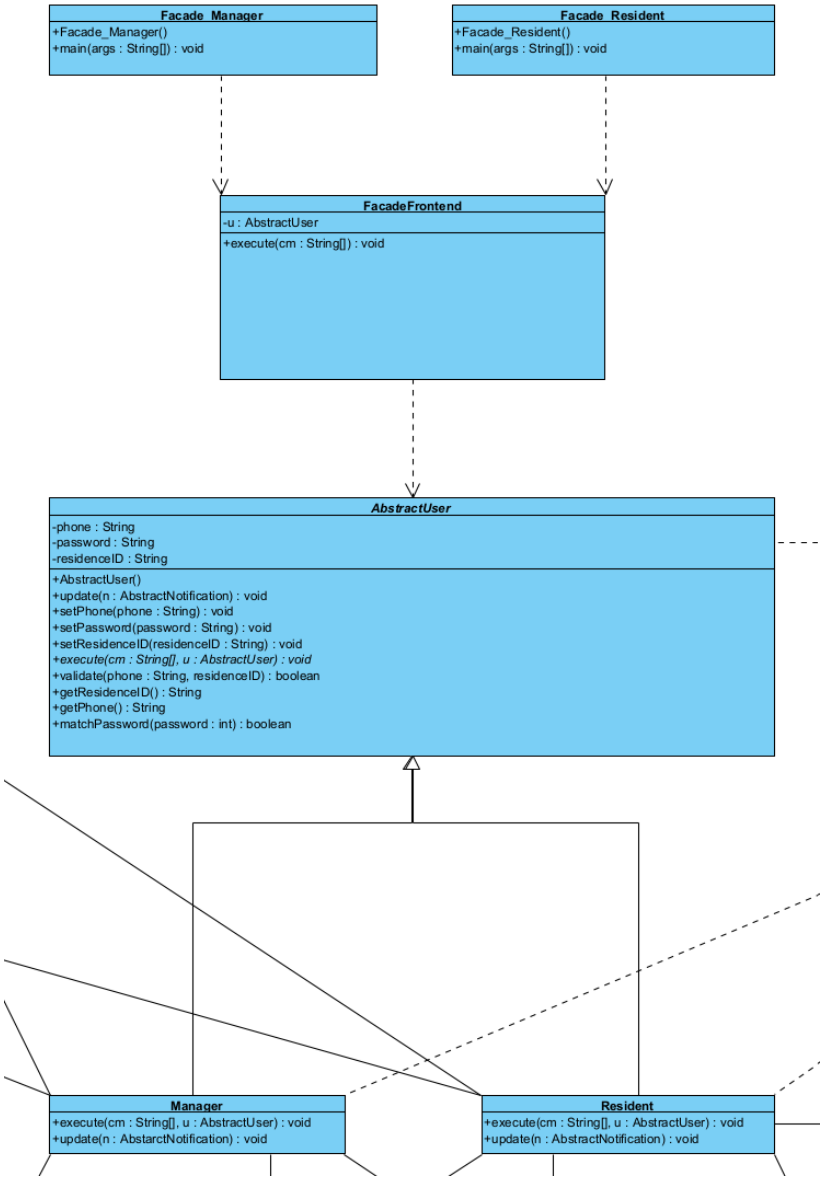

Figure 14: Facade

In the provided class diagram, the Facade pattern is represented by the **FacadeFrontend** class. This class acts as a facade for the operations clients such as **Facade_Manager** and **Facade_Resident** can perform.

- **FacadeFrontend**: The facade provides simple access methods like **execute()**. It abstracts the complexities of the underlying subsystem, which is represented by **AbstractUser** and its concrete implementations.
- **AbstractUser**: Defines a standard interface for different types of users. It contains methods like **update()**, **setPhone()**, **setPassword()**, and so on that are common across different user types.
- **Manager and Resident**: Extend from **AbstractUser**, representing specific user roles with behaviors and attributes tailored to their needs. They override the **execute()** method to perform their specific actions.
- **Relationships**:

  (1) The **Facade_Manager** and **Facade_Resident** classes depend on the **FacadeFrontend** to execute commands. They do not interact directly with the **Manager** or **Resident** classes, thus adhering to the Facade pattern.

  (2) The **FacadeFrontend**, in turn, delegates the client requests to the appropriate methods of **AbstractUser**, which are implemented by the **Manager** or **Resident**.

  (3) The dashed lines represent the use relationships, indicating that the **FacadeFrontend** uses instances of **AbstractUser** to fulfill the requests.

The use of the Facade pattern in this app simplifies client interaction with the system's backend, thereby providing an encapsulated, high-level interface that hides the subsystems' complexity from the end user.

4.3.2 Singleton Pattern

The Singleton pattern ensures a class has only one instance and provides a global point of access to it. This pattern is particularly useful for controlling access to resources that are shared across various parts of an application.

Characteristics of this pattern are as follows:

- **Private Constructor**: Prevents direct construction calls with the new operator.
- **Private Static Instance**: A static member that holds the sole instance of the class, ensuring controlled access.
- **Public Static Access Method**: The getInstance() method provides a way to access the unique instance of the class.

It is utilized within our community service app to manage centralized services that include user management, notification dispatching, maintenance requests, reservation handling, and complaint processing. Implementing these services as singletons guarantees that there is a single, coherent state and access point throughout the application's lifecycle.

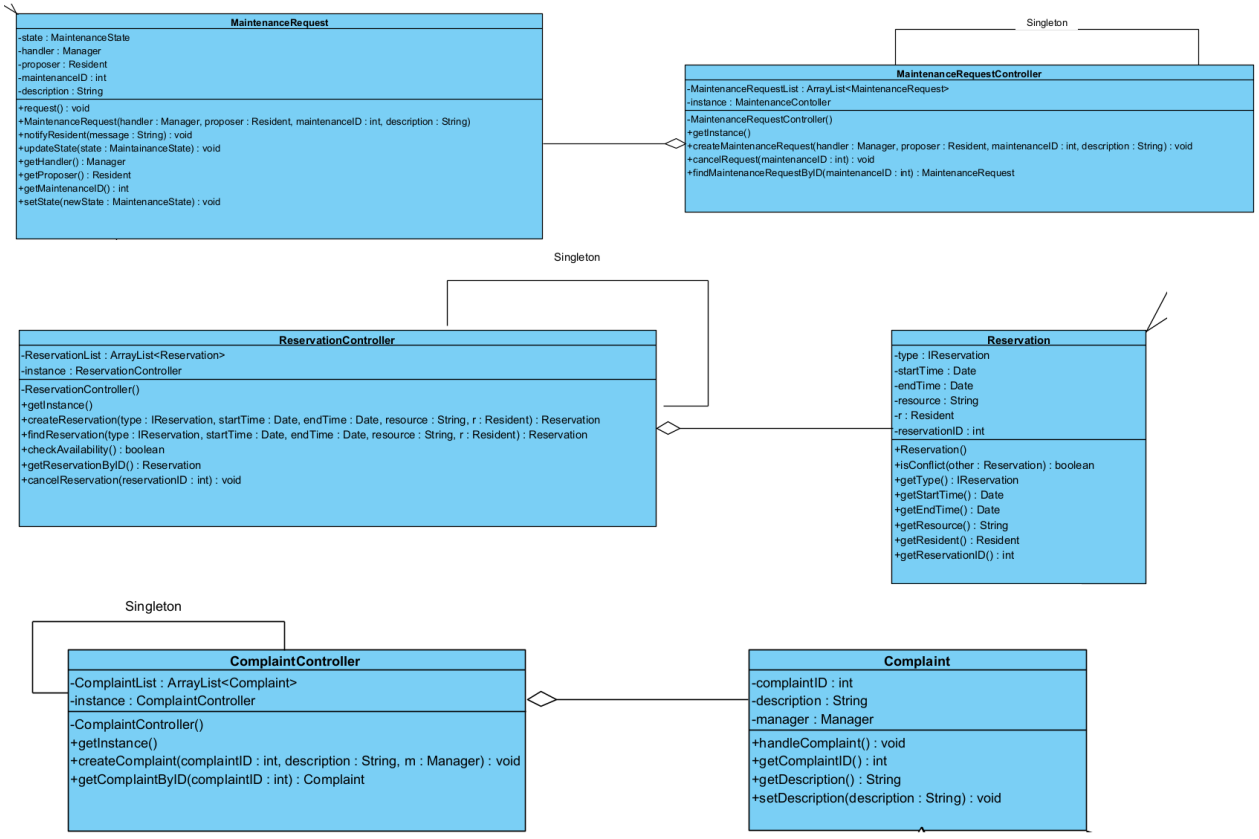

Figure 15: Examples of Singleton pattern

For example, UserController maintains a list of AbstractUser objects, providing a central management point for user information and actions. It interfaces with various client classes that represent users of the system, managing their authentication and interaction with the system's resources.

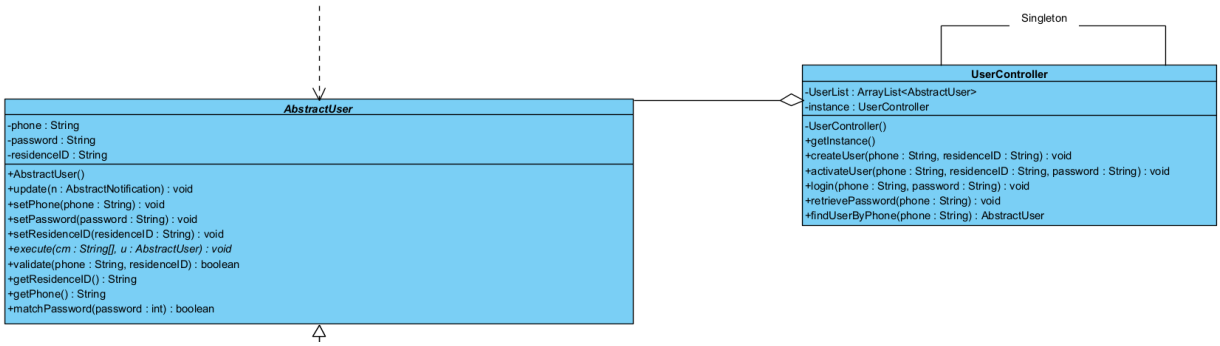

Figure 16: Singleton pattern for UserController

By employing the Singleton pattern, the application's core management functionalities are provided in a consistent and controlled manner. This architectural decision aids in reducing overhead, preventing inconsistent states, and ensuring that these core components are readily available throughout the application without the need for multiple instantiations.

4.3.3 Command Pattern

The Command pattern encapsulates a request as an object, thereby allowing for the parameterization of clients with queues, requests, and operations. It provides the flexibility to issue commands without knowing anything about the operation being performed or the receiver of the command. The key components of the Command pattern are:

- **Command Interface**: Declares an interface for executing operations.
- **Concrete Command**: Defines a binding between a Receiver object and an action.
- **Client**: Creates a Concrete Command object and sets its receiver.
- **Invoker**: Asks the command to carry out the request.
- **Receiver**: Knows how to perform the operations associated with carrying out a request.

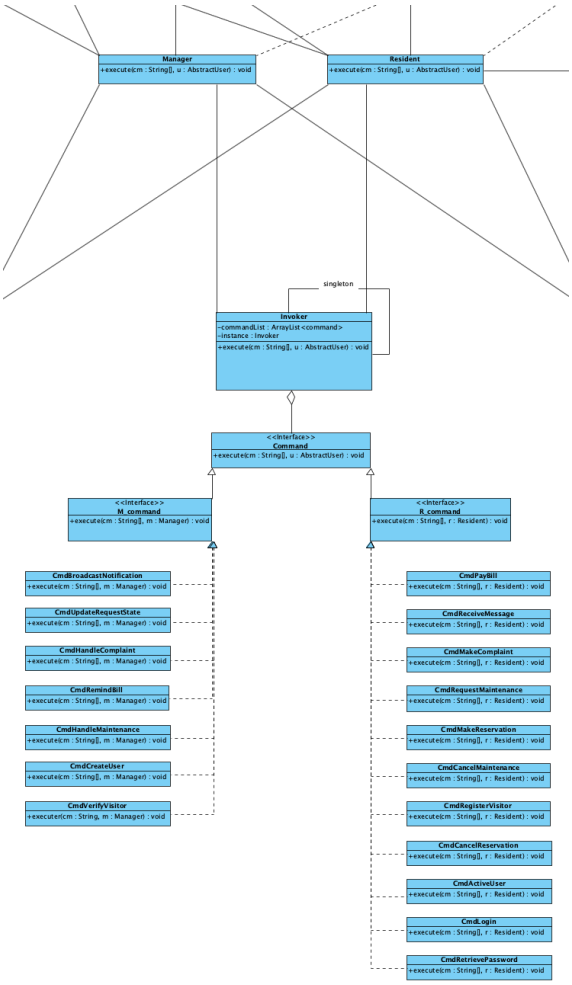

Figure 17: Command Pattern

Our class diagram effectively demonstrates the implementation of the Command pattern within our system's architecture, providing a flexible and extensible framework for executing user actions.

- **Command Interface**: Our Command interface defines the executable operation, which in our case is the execute method that accepts a command string and an **AbstractUser** as arguments and capitalizes on polymorphism that is designed to be overridden by concrete command classes tailored to specific actions related to Resident and Manager entities.
- **Concrete Command**: We have multiple concrete command classes such as **CmdPayBill**, **CmdReceiveMessage**, **CmdMakeComplaint**, each implementing the Command interface for specific actions pertaining to the Resident user type. Similarly, commands like **CmdBroadcastNotification** and **CmdUpdateRequestState** are

designed for Manager operations.

- **Client**: In our context, Manager and Resident classes act as clients. They are aware of the commands but do not execute them directly. Instead, they pass the commands to the invoker.
- **Invoker**: The Invoker class, implemented as a singleton to ensure a single point of command dispatch within the system, maintains a list of commands and forwards the execute method calls to the corresponding concrete commands.
- **Receiver**: While the class diagram does not explicitly identify receivers, they are implicit within the concrete command implementations. Each concrete command knows which receiver to act upon to fulfill the command. For instance, **CmdPayBill** would know which subsystem or method to invoke to process a payment.

The design allows for new commands to be added with minimal changes to existing code, showcasing the Command pattern's support for the Open/Closed Principle. It provides a flexible and scalable solution for extending the functionality of our system to accommodate future requirements.

4.3.4 Observer Pattern

The Observer Pattern plays a critical role in our system's notification mechanism. It facilitates a publish-subscribe model, allowing for a clean separation between the state management of user notifications and disseminating notification information to various subscribers.

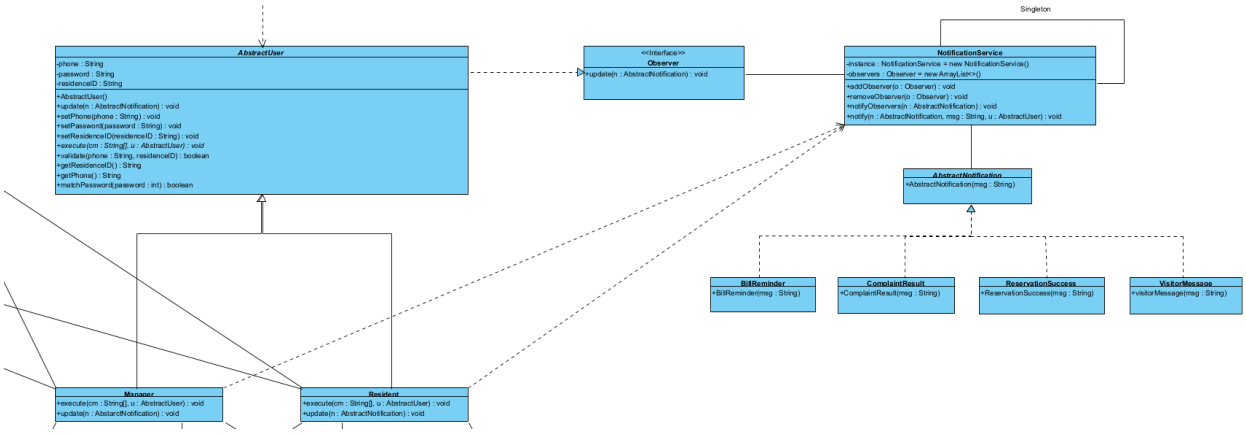

Figure 18: Observer Pattern

The class diagram presents the following structure and dynamics in the Observer pattern's

implementation:

- **Observer Interface**: Outlines the protocol for receiving updates from subjects. In our system, the Observer interface declares an update function triggered whenever a noteworthy event occurs.
- **AbstractUser**: This entity represents the base class for different user roles that may observe changes. Concrete implementations, such as Managers and Residents, are equipped with the ability to receive and process notifications.
- **NotificationService**: Serves as the central hub for managing notification subscriptions and broadcasts. Operating under the Singleton pattern, it ensures a unified point for all notification activities within the system.
- **Concrete Observers**: Entities that subscribe to the **NotificationService** and implement the Observer interface. When the notification state changes, these observers receive updates, allowing for reactive behaviors specific to their roles.

When the state of **NotificationService** changes, it automatically invokes the **notifyObservers** method. This method iterates through the registered observers and calls their update method, passing along the pertinent notification data encapsulated within **AbstractNotification**.

Implementing the Observer Pattern in our notification infrastructure provides an efficient mechanism to broadcast state changes to interested parties. This pattern enhances the system's flexibility and responsiveness, which is essential in modern service-oriented architecture.

4.3.5 Strategy Pattern

The Strategy pattern is a behavioral pattern that enables selecting an algorithm's behavior at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions on which in a family of algorithms to use.

The intent behind the Strategy pattern in our system is to allow the Payment class to be flexible in terms of the payment calculation and execution methods without altering its structure. It defines a family of algorithms, encapsulates each one of them, and makes them interchangeable. The Payment class's behavior can change dynamically depending on the payment scenario and the payment method chosen by the user.

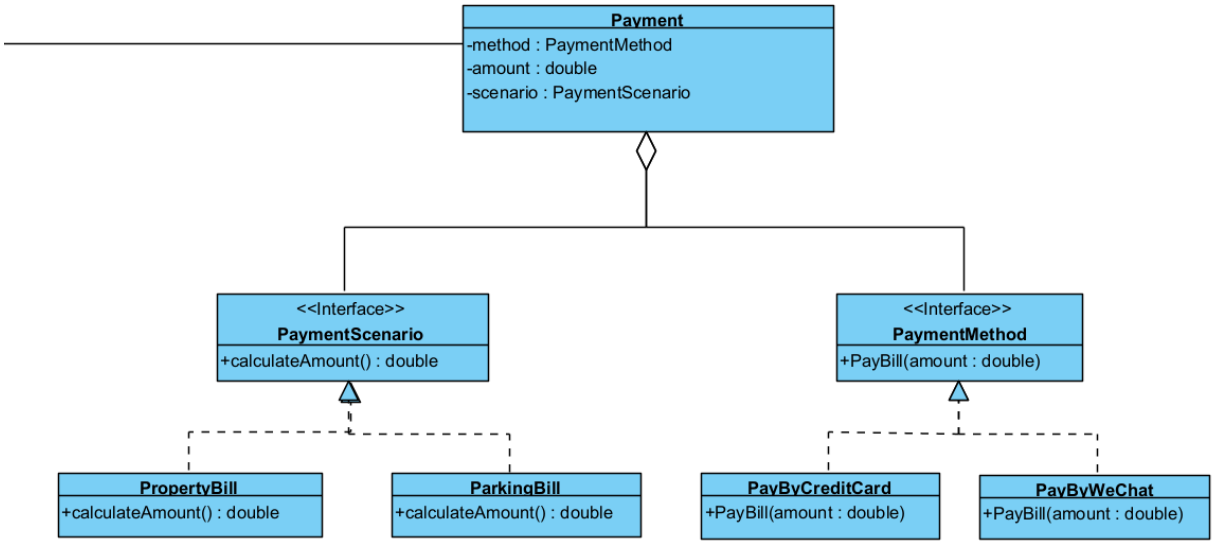

Figure 19: Strategy Pattern

The class diagram presents the following components in the Strategy pattern's implementation:

- **Payment**: A class that maintains a reference to a **PaymentMethod** and a **PaymentScenario** instance.It uses a diamond symbol to denote composition, indicating it's composed of strategy instances.
- **PaymentMethod**: An interface defining the **PayBill(amount: double)**method. Concrete strategies like **PayByCreditCard** and **PayByWeChat** implement this interface, providing specific payment implementations.
- **PaymentScenario**: An interface with the **calculateAmount()**: double method that is implemented by different scenarios like **PropertyBill** and **ParkingBill**.These concrete strategies calculate the payment amount based on different billing scenarios.

The Strategy pattern in the payment system provides the necessary flexibility to handle various payment methods and calculation scenarios. This design allows easy expansion and modification of payment logic, making the system more adaptable to future changes.

4.3.6 State Pattern

The State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

The purpose of applying the State pattern within our maintenance system is to enable the MaintenanceRequest objects to behave differently depending on their state, which can be one of the following: Cancelled, Completed, InProgress, or Pending. This pattern eliminates the need for large conditional statements and allows for a cleaner design where state-specific behaviors are encapsulated within state objects.

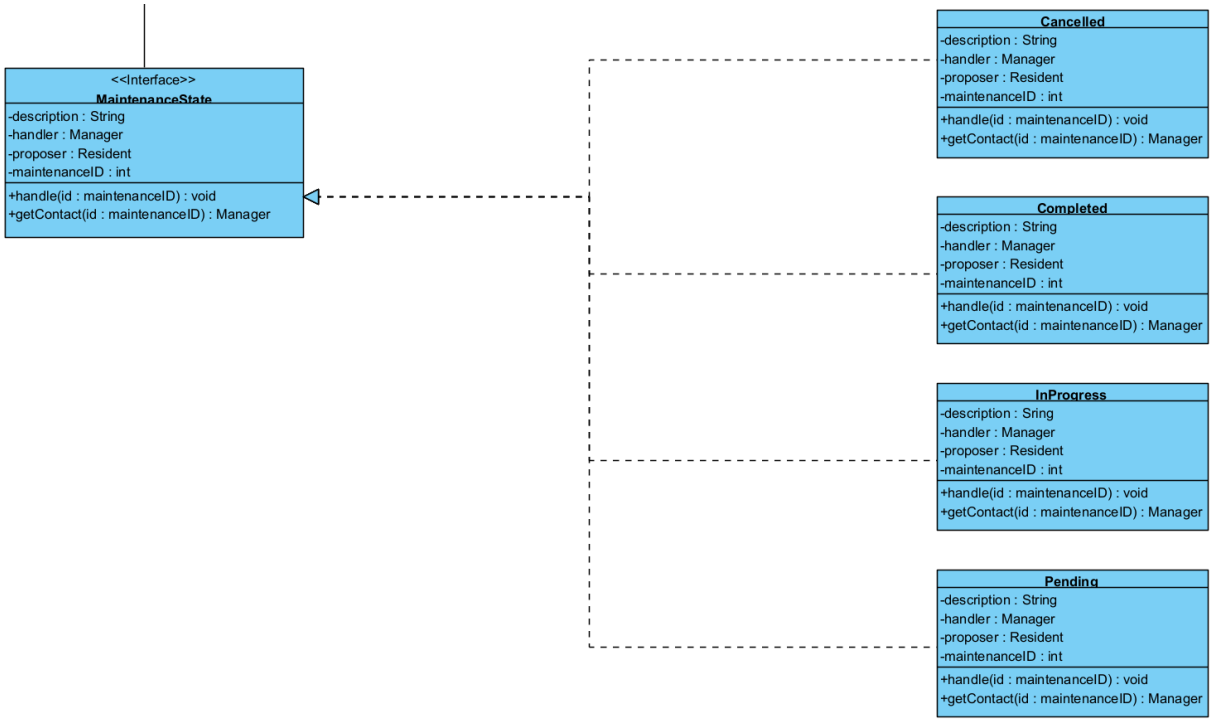

Figure 20: State pattern

The provided class diagram shows the following elements, which are crucial for the State pattern implementation:

- **MaintenanceState**: An interface that defines the contract for encapsulating the behavior associated with a particular state of the **MaintenanceRequest**. Methods like **handle()** and **getContact()** define actions and inquiries related to the maintenance request.
- **Concrete States (Cancelled, Completed, InProgress, Pending)**: These classes implement the **MaintenanceState** interface, providing specific implementations of

the **handle()** and **getContact()** methods. Each concrete state class encapsulates the state-specific behavior for a maintenance request.

The State pattern provides a robust structure for managing maintenance requests in our system. It offers a systematic and flexible way to accommodate new states and behaviors associated with the lifecycle of a maintenance request, ensuring the system is scalable and easy to maintain.

4.3.7 Factory Method Pattern

In the design of our complaint handling framework, the Factory Method Pattern enables the dynamic creation of complaint objects. It provides a mechanism for encapsulating the instantiation logic and delegating the object creation responsibility to specialized factory classes.

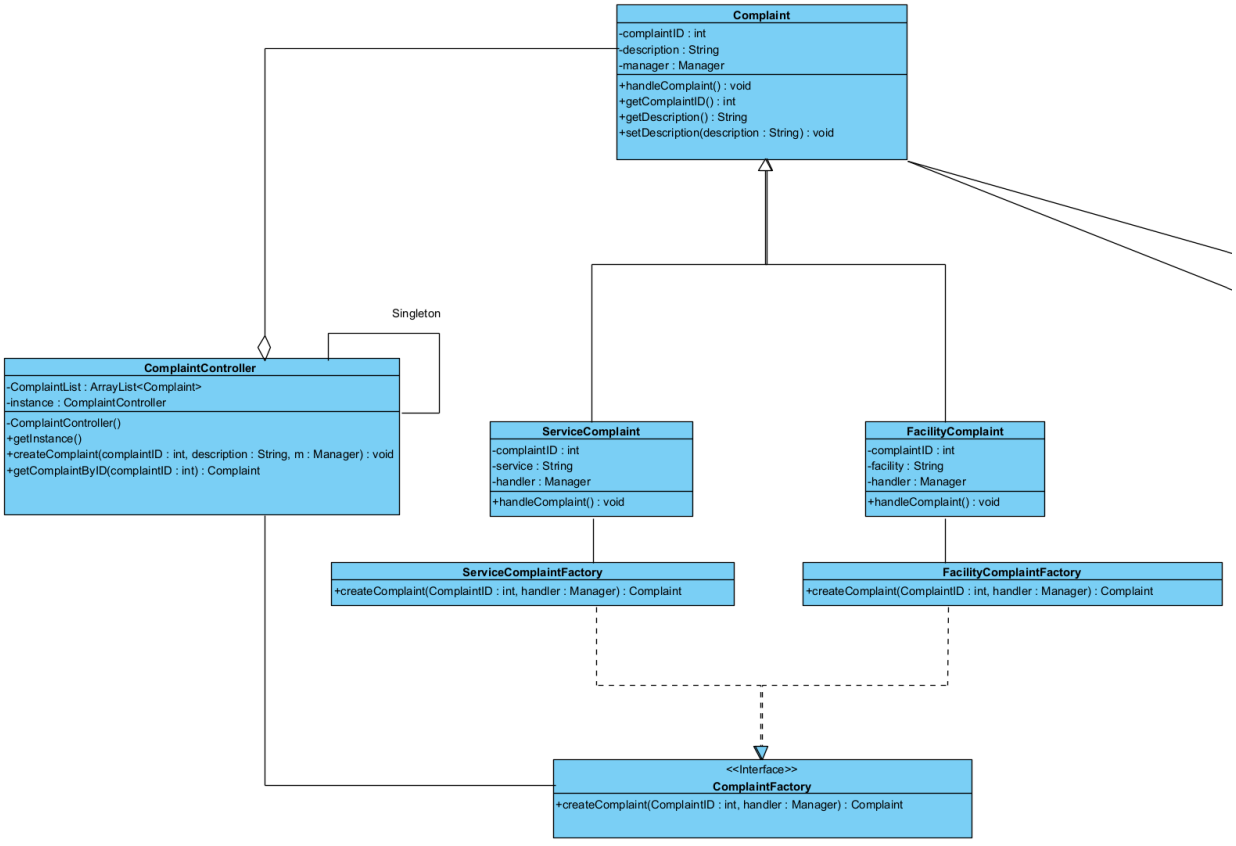

Figure 21: Factory pattern

The Factory Method Pattern offers a way to decouple client code from the concrete classes that need to be instantiated. This abstraction is crucial for our system, which requires flexibility in managing various types of complaints—each with distinct attributes and handling procedures.

- **ComplaintController**: Operates as a Singleton, ensuring a single point of reference for managing complaints, thereby coordinating the overall complaint registration and resolution process.
- **Complaint**: Serves as the product interface in the Factory Method Pattern lexicon. It outlines a common framework for all complaints, providing consistency across different complaint resolutions.
- **Concrete Complaint Classes**: Such as **ServiceComplaint** and **FacilityComplaint**, represent specific complaint categories. Each inherits from the Complaint superclass and implements particular handling strategies.
- **Factory Classes**: Including **ServiceComplaintFactory** and **FacilityComplaintFactory**, these play the role of creator in the Factory Method Pattern. They override the factory method to spawn instances of Complaint subclasses, thus encapsulating the creation logic. Utilizing factory classes for complaint generation allows the **ComplaintController** to remain agnostic of the concrete complaint types. This results in a more maintainable and scalable system, as new types of complaints can be added with minimal changes to the existing codebase.

In conclusion, the adoption of the Factory Method Pattern within our complaint handling architecture provides a robust and scalable approach to object creation. It supports the growth and evolution of our service offerings, simplifying the integration of new complaint types as the system expands

5 Sequence Diagram Design

5.1 Account Module

There are 3 **Resident** commands and 1 **Manager** command in the Account module. The **Resident** commands are **CmdActivateUser**, **CmdUserLogin**, and **CmdRetrievePassword**, while the **Manager** command is **CmdCreateUser**.

5.1.1 Create User

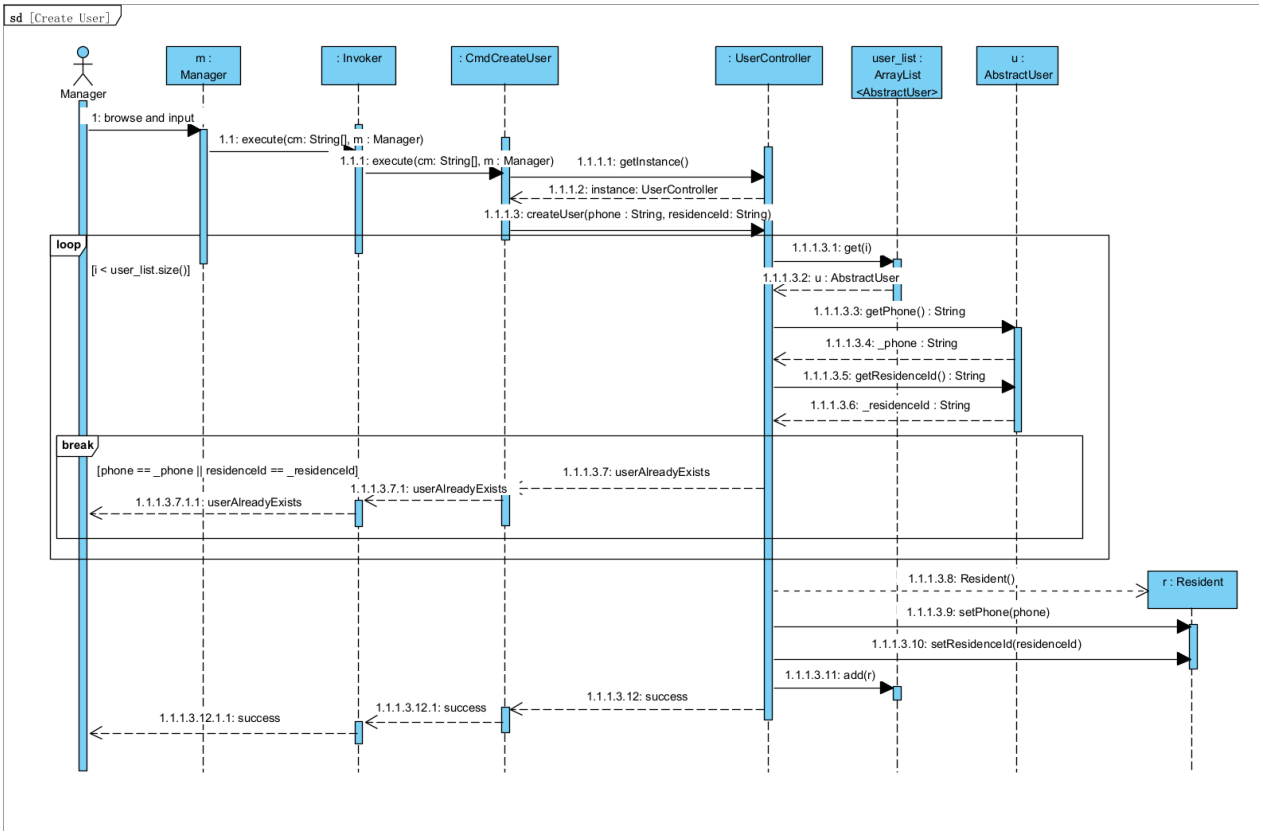

Figure 22: Create User

The command pattern is used throught the Community Service & Management System to handle various requests from both the residents and the managers. User requests will be prompted through the **Invoker** class, which will then call the appropriate command.

The **CmdCreateUser** class is a **Manager** command that creates a new user. When a new resident is registered, the manager will create a new user account for each of them. The manager will input the user's phone and residence ID, which is a global unique identifier generated by the system, and confirm the user's registration.

Internally, the manager's **execute()** method will call the **Invoker** class, which then forwards the request to the **CmdCreateUser** class. The command class then invokes the **createUser()** method within the **UserController** class, a Singleton class that handles all user-related operations.

The **createUser()** method will iterate through all existing users to check if the new user's phone number or residence ID already exists.

There are two possible outcomes:

- If either the phone number or residence ID already exists, the method will return an **User Already Exists** error message, which is eventually displayed to the manager.
- Otherwise, a new **Resident** object is created with the provided phone number and residence ID, and appended to the **user_list** in **UserController**, and the method will return a success message.

5.1.2 Activate User

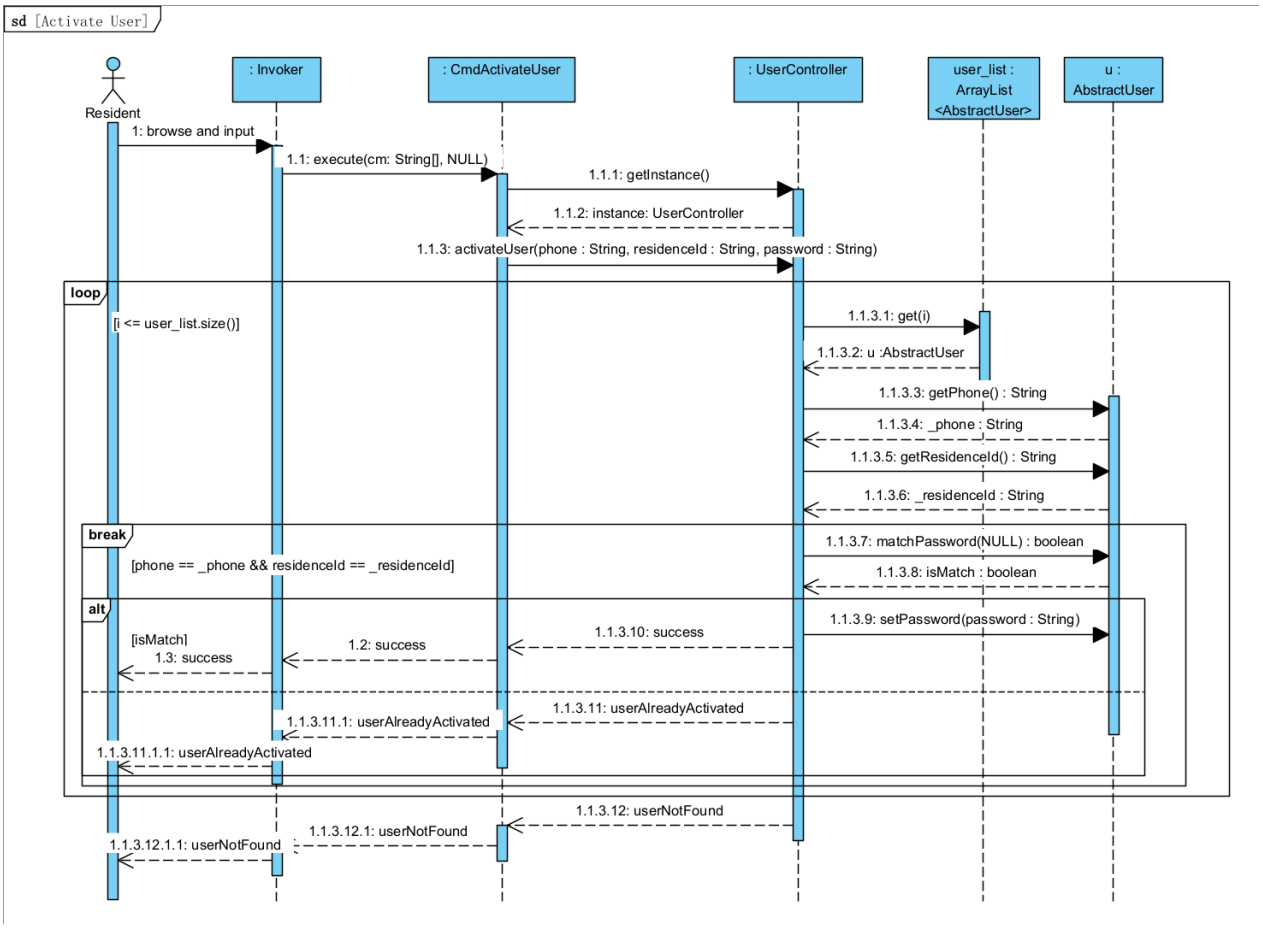

Figure 23: Activate User

The **CmdActivateUser** class is a **Resident** command that activates a user after it is created by the manager. After entering the phone, residence ID, and the password, the request will be sent to the **Invoker** and then to the **CmdActivateUser** class. Since at this point, no login session has been created, the **execute()** method will use **NULL** as the user parameter.

CmdActivateUser calls the **activateUser()** method in the **UserController** class, which will iterate through all users to find if a user with both the phone number and residence ID matches the provided information.

There are three possible outcomes:

- If the user is found, the method will check if the user is already activated, by checking if the user has set a password:
  - If the user is not activated, given by the fact that the password is **NULL**, the

method will set the password, which uses a hashing algorithm to store the password securely, and return a success message.

– If the user is found but already activated, the method will return an **User already activated** error message.

In either case, the method will break the user iteration loop and immediately return the message to the **CmdActivateUser** class.

- If the user is not found, the method will return an **User not found** error message.

5.1.3 User Login

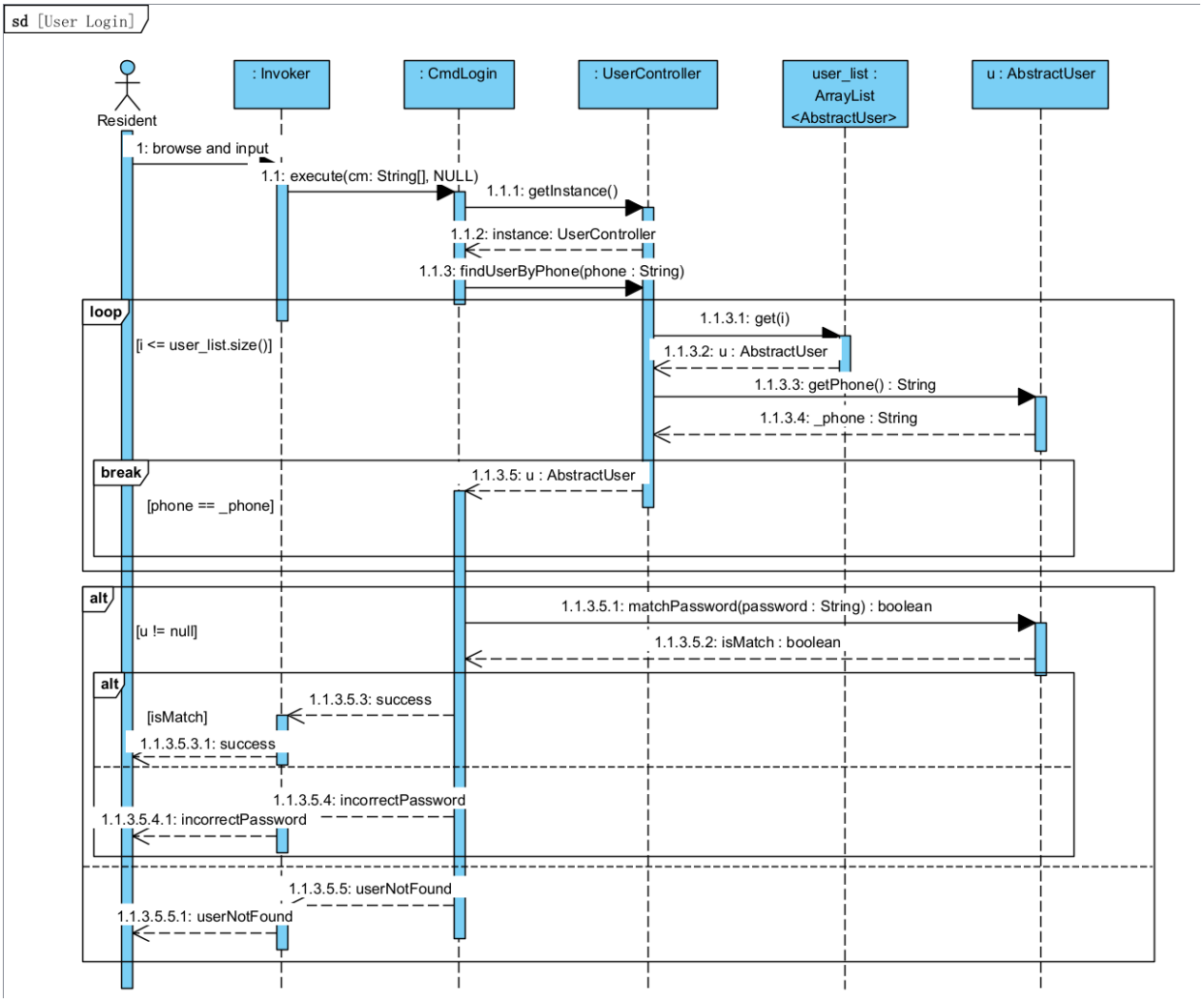

Figure 24: User Login

The **CmdUserLogin** class is a **Resident** command. After entering the phone and the password, the request will be sent to **Invoker** and **CmdUserLogin**.

CmdUserLogin calls the **findUserByPhone()** method in the **UserController** class, which returns the user object with the given phone number, or **NULL** if the not found.

There are three possible outcomes:

- If the user is found, **CmdUserLogin** calls **matchPassword()** method of the returned user object, which will compare the hashed password with the hash of the provided password:
  - If the password matches, the method will return a success message.
  - If the password does not match, the method will return an **Incorrect password** error message.
- If the user is not found, the method will return an **User not found** error message.

5.1.4 Retrieve Password

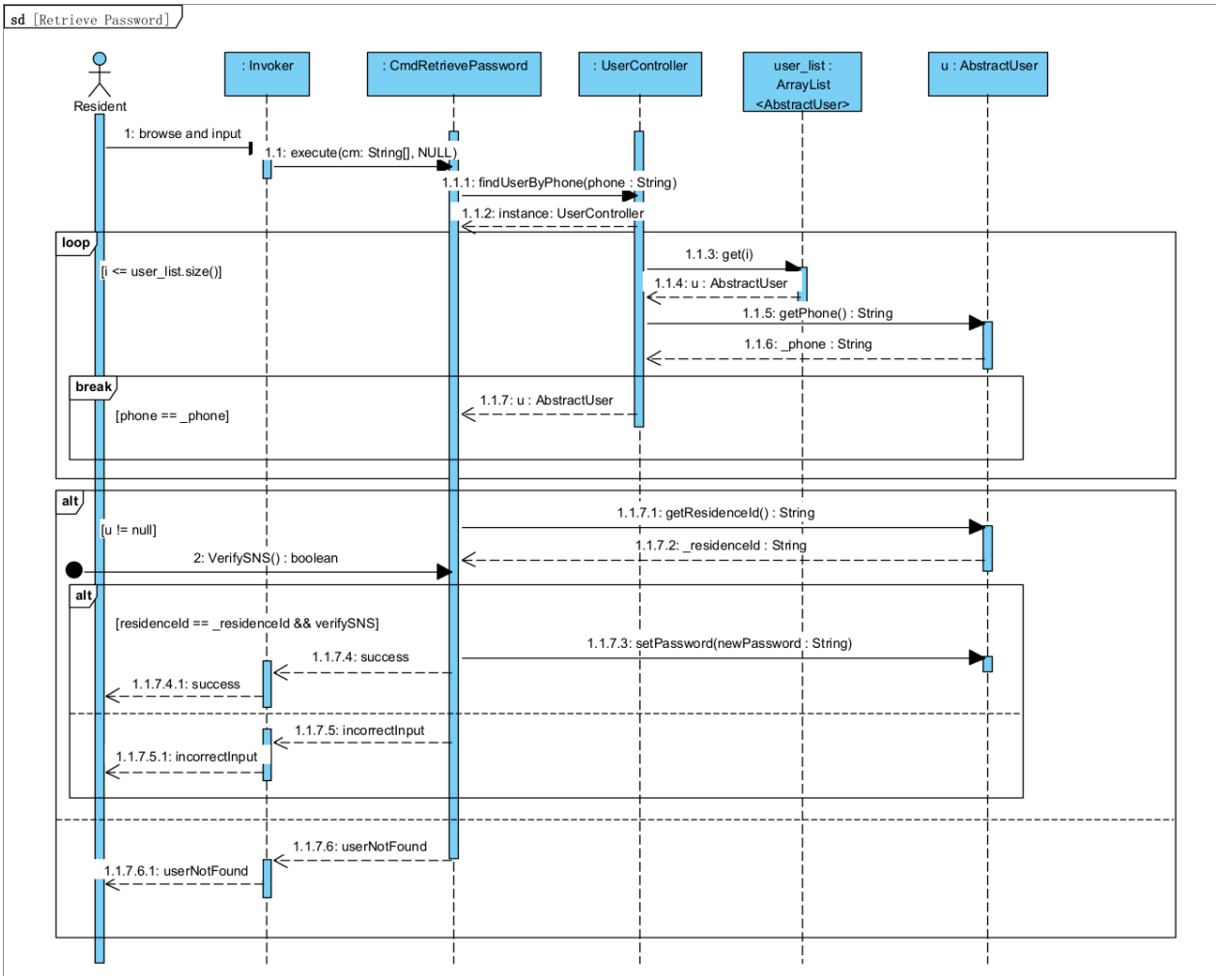

Figure 25: Retrieve Password

The **CmdRetrievePassword** class is a **Resident** command. If the user forgets their password, they can retrieve it by entering their phone number, residence ID and a new password. The request will be sent to **Invoker** and **CmdRetrievePassword**.

CmdRetrievePassword calls the **findUserByPhone()** method first. If the user is found, the user will be prompted to enter a SMS verification code, which is sent to the user's phone number. The result is given by an external boolean-value function, **verifySMS()**.

There are three possible outcomes:

- If the user is found, and both the residence ID and the SMS verification code are correct, the method will call the **setPassword()** method of the user object to set the new password, and return a success message.

- If the user is found, but either the residence ID or the SMS verification code is incorrect, the method will return an **Incorrect input** error message.
- If the user is not found, the method will return an **User not found** error message.

5.2 Visitor Module

There is 1 **Resident** and 1 **Manager** command in the Visitor module, which are **CmdRegisterVisitor** and **CmdVerifyVisitor** respectively.

5.2.1 Register Visitor

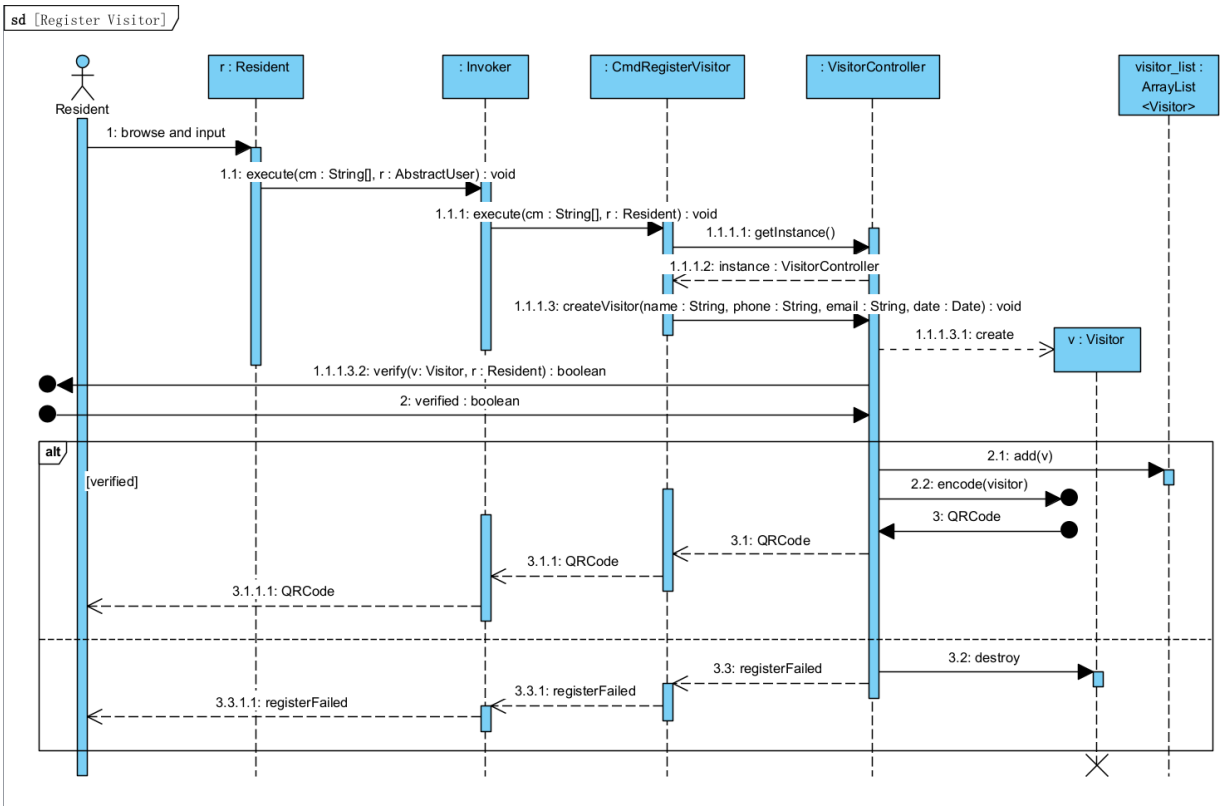

Figure 26: Register Visitor

The **CmdRegisterVisitor** class is a **Resident** command. After inputing the visitor's name, phone number, email and the date and time of the visit, the request will be sent to **Invoker** and **CmdRegisterVisitor**.

The **CmdRegisterVisitor** class calls the **createVisitor()** method in the **VisitorController** class. It creates a **Visitor** object with the provided information, and also informs the manager to approve the visit via **verify()** method (see **CmdVerifyVisitor**).

There are two possible outcomes:

- If the manager approves the visit, the **Visitor** object will be appended to the **visitor_list**. The visitor information will be encoded into a QR code and sent to the resident.
- If the manager rejects the visit, the **Visitor** object will be discarded and **Register Failed** message will be returned to the resident.

5.2.2 Verify Visitor

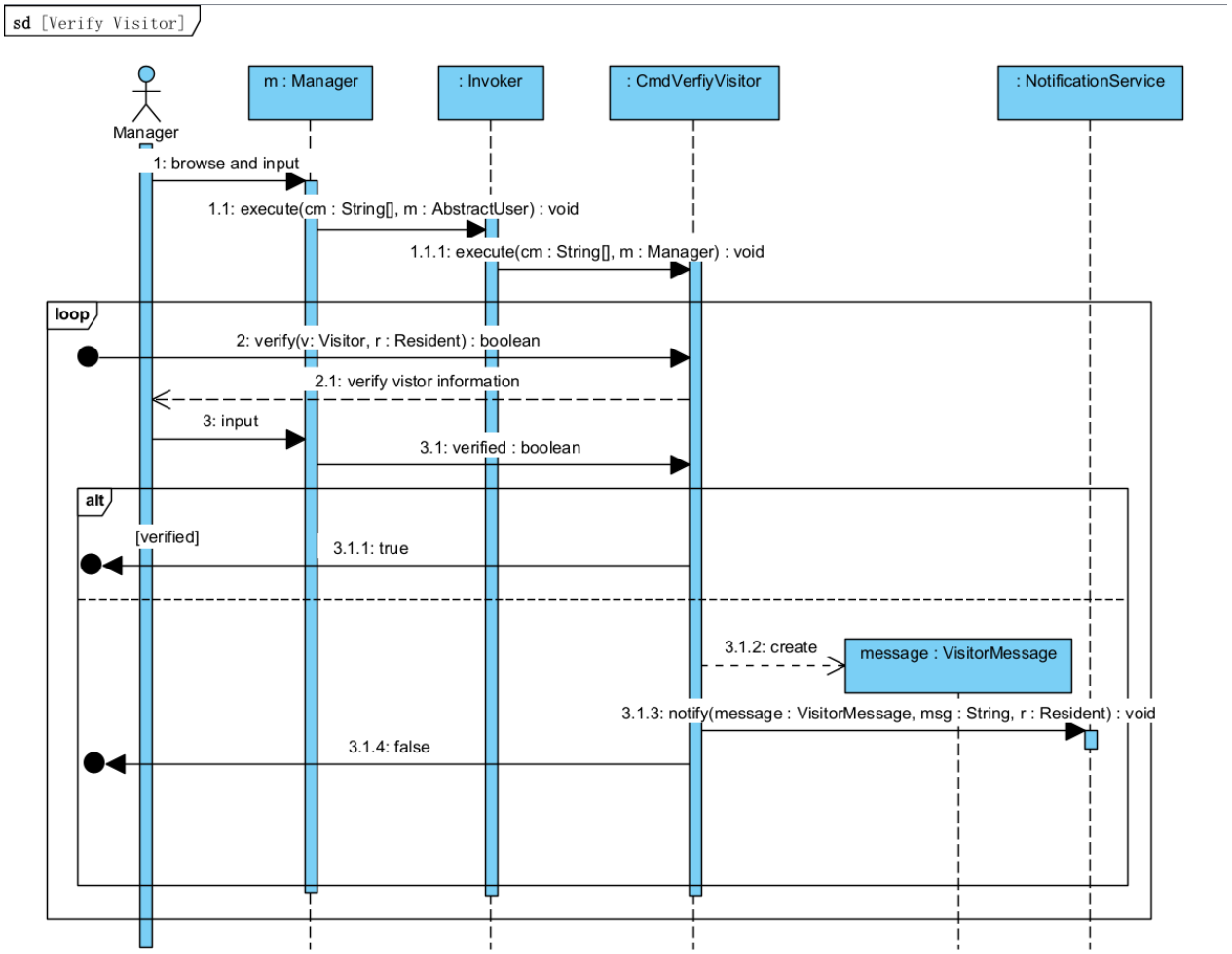

Figure 27: Verify Visitor

The **CmdVerifyVisitor** class is a **Manager** command.

As mentioned in the previous sequence diagram, when a visitor is registered, the **verify()** method is called with the **Visitor** object and the corresponding **Resident** who registered

the visitor. If there are any visitors waiting for approval, the system will display the visitor information and relevant details to the manager. The manager can then choose to approve or reject the visit.

If the manager approves the visit, the function simply returns **true**, and the user-side application will deal with the rest of the process. If the manager rejects the visit, the function sends a **VisitorMessage** to the resident, informing them that the visit has been rejected, and returns **false**.

5.3 Payment Module

There is 1 **Resident** command in the Payment module, which is **CmdPayBill**.

5.3.1 Pay Bill

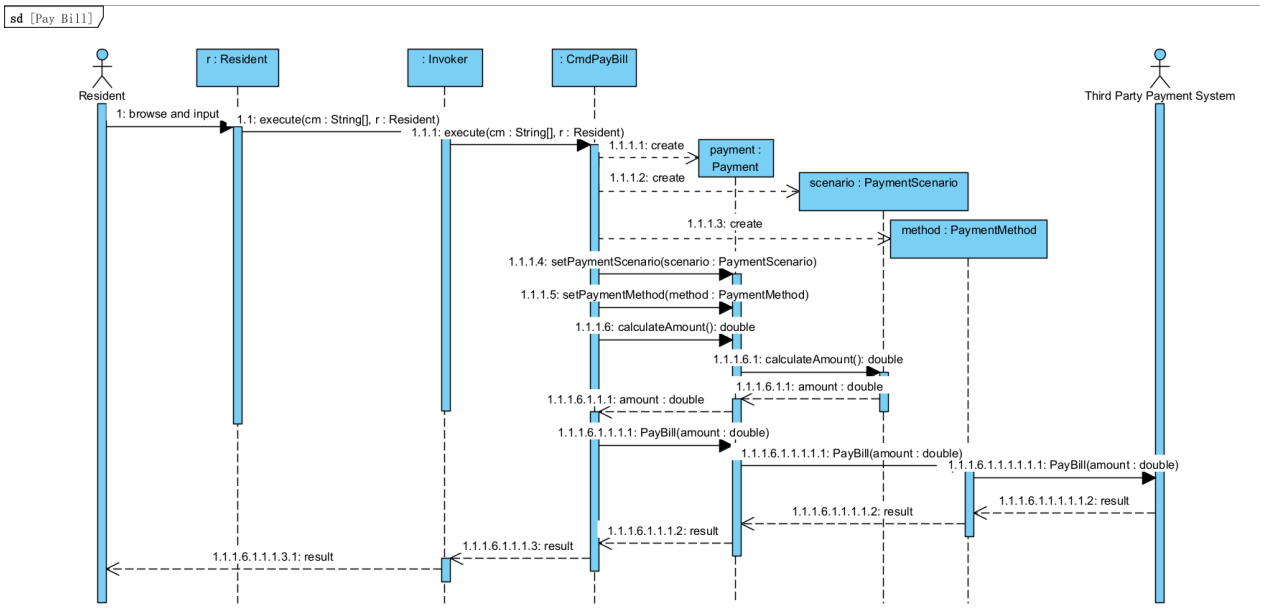

Figure 28: Pay Bill

After the resident receives a bill reminder, they can pay the bill by choosing the type of bill (like property fee, parking fee) and the payment method (like credit card, WeChat Pay). The request will be sent to **Invoker** and **CmdPayBill**.

CmdPayBill will create a **Payment** object with corresponding **PaymentScenario** and **PaymentMethod** objects as fields.

Then, it invokes **calculateAmount()** method of the **Payment** object to calculate the amount to be paid. Since different types of bills may have different calculation methods, polymorphism is used to implement the **calculateAmount()** method in different subclasses of **PaymentScenario**.

Finally, it invokes **PayBill()** method of the **PaymentMethod** object, which interacts with **Third Party Payment System** to complete the transaction. The transaction result will be returned to the resident.

5.4 Notification Module

Notify User is an internal service used by various modules, including **Visitor**, **Bill**, **Reservation** and **Complaint** modules. It is used to send notifications to specific users.

5.4.1 Notify User

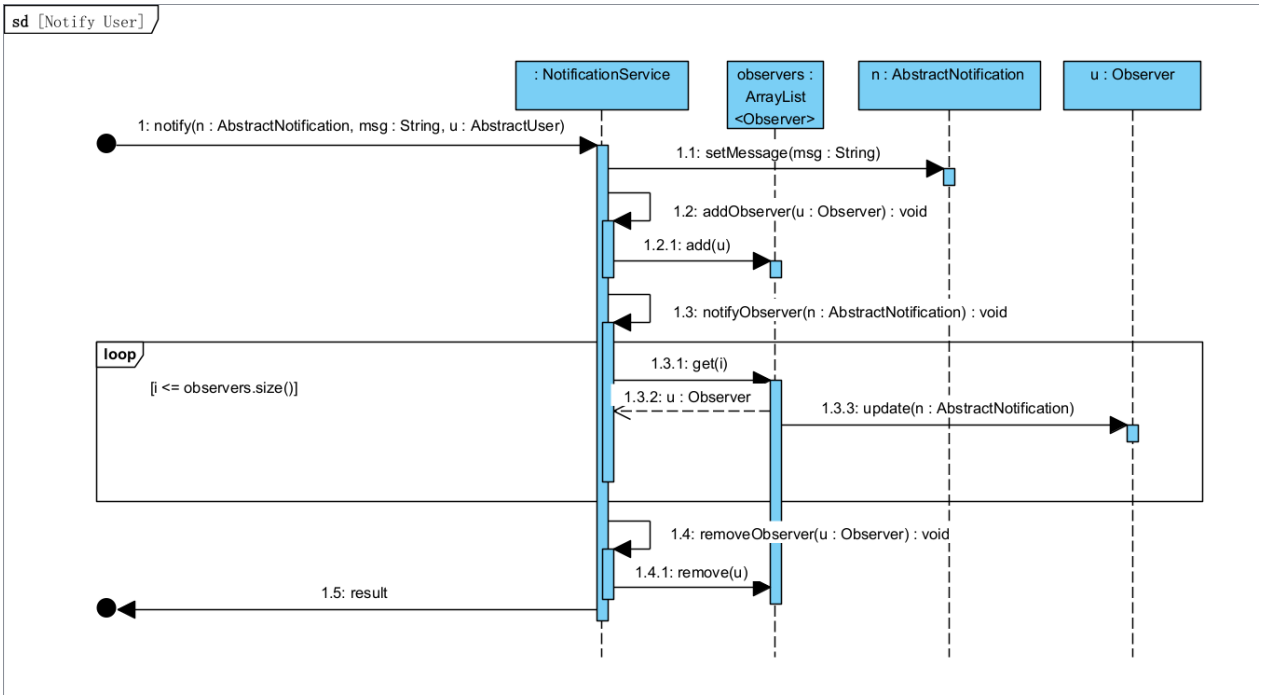

Figure 29: Notify User

When another module calls **notify()** method of the **NotificationController** class, it must provide the type and the content of the notification, as well as the user to be notified.

The process can be divided into three steps:

1. **addObserver()**: **NotificationService** adds the user to the observer list.
2. **notifyObserver()**: **NotificationService** iterates through the observer list and sends the notification to the user, by calling their **update()** method.
3. **removeObserver()**: After the notification is sent, the user is removed from the observer list.

5.5 Maintenance Module

There are 2 **Resident** commands and 1 **Manager** command in the Maintenance module. The **Resident** commands are **CmdRequestMaintenance** and **CmdCancelMaintenance**, while the **Manager** command is **CmdUpdateRequestState**.

5.5.1 Request Maintenance

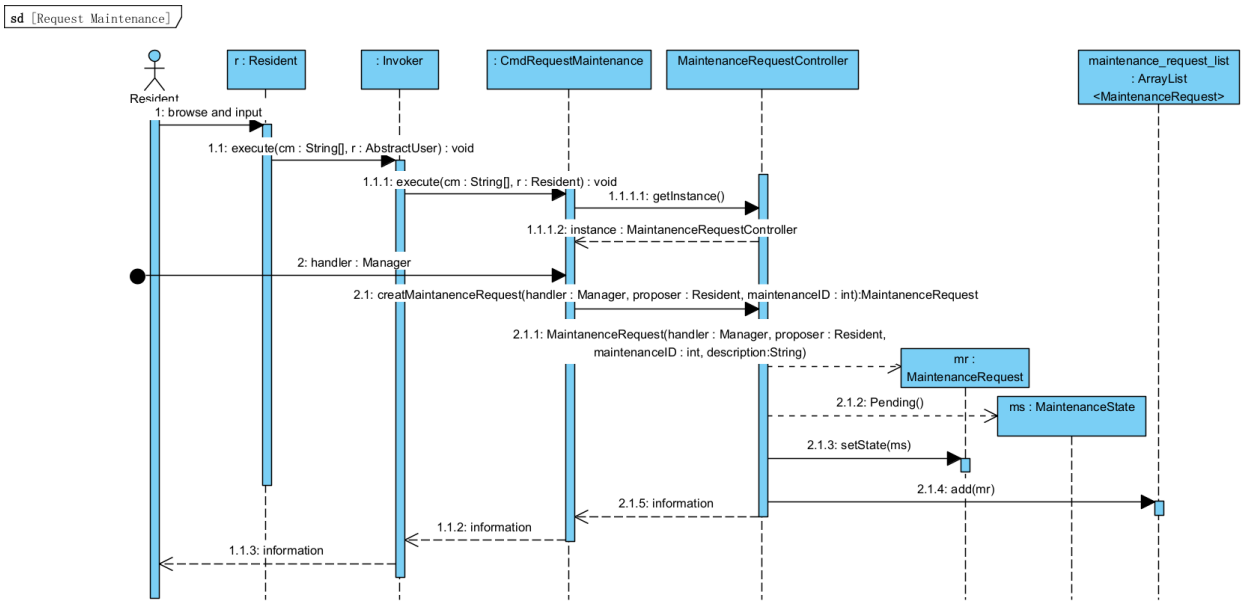

Figure 30: Request Maintenance

The **CmdRequestMaintenance** class is a **Resident** command. After inputing the maintenance type and description, the request will be sent to **Invoker** and **CmdRequestMaintenance**.

Each request will be assigned a handler. **CmdRequestMaintenance** calls the **CreateMaintenanceRequest()** method in the **MaintenanceRequestController** class, which creates a **MaintenanceRequest** object with the provided information, including the handler. Its state will be initialized as **Pending**. Then, the request will be appended to the **maintenance_request_list**.

Finally, it returns relevant information (including the maintanence ID and the handler) to the resident.

5.5.2 Cancel Maintenance

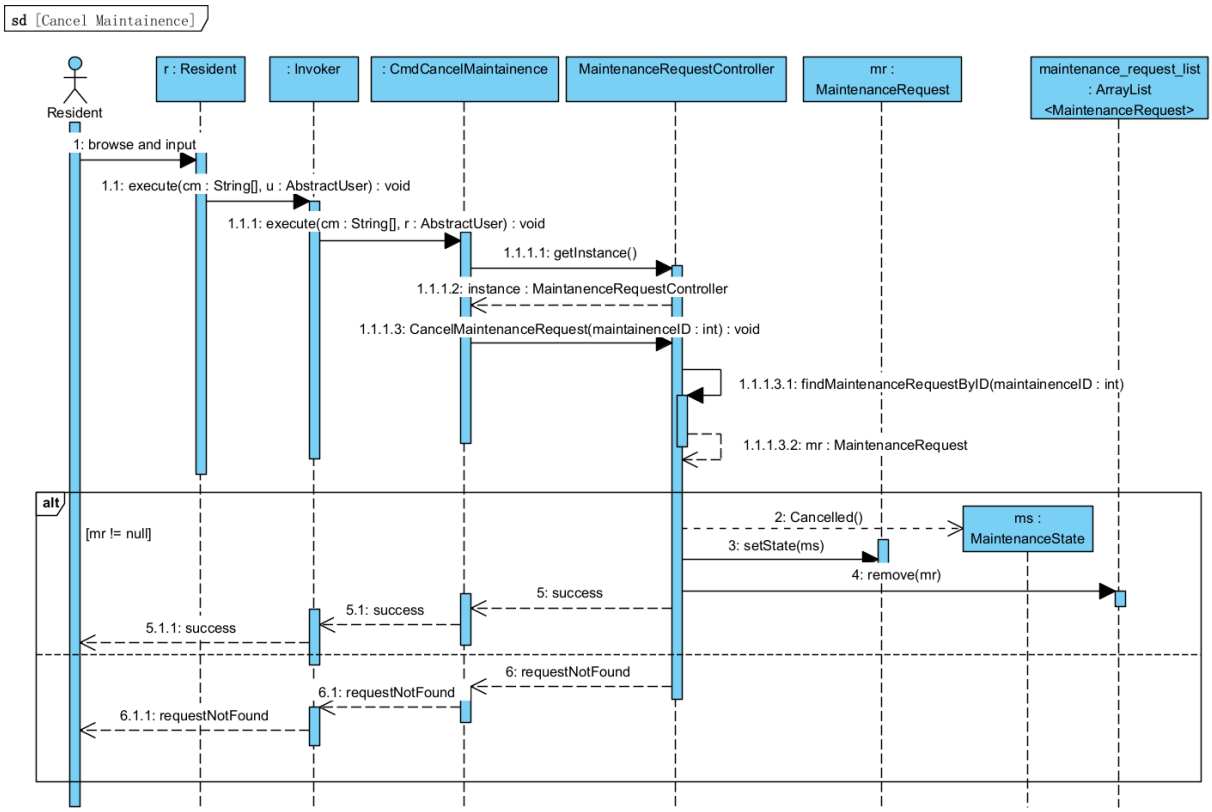

Figure 31: Cancel Maintenance

The **CmdCancelMaintenance** class is a **Resident** command. In the event that the resident wants to cancel a maintenance request, they can choose the request to cancel. Required information includes the maintenance ID.

CmdCancelMaintenance calls the **CancelMaintenanceRequest()** method in the **MaintenanceRequestController** class, which will find the request with the given maintenance ID by **findMaintenanceRequestByID()** method.

If the request is found, the method will update the request state to **Cancelled**, remove it from the list and discard it.

5.5.3 Update Request State

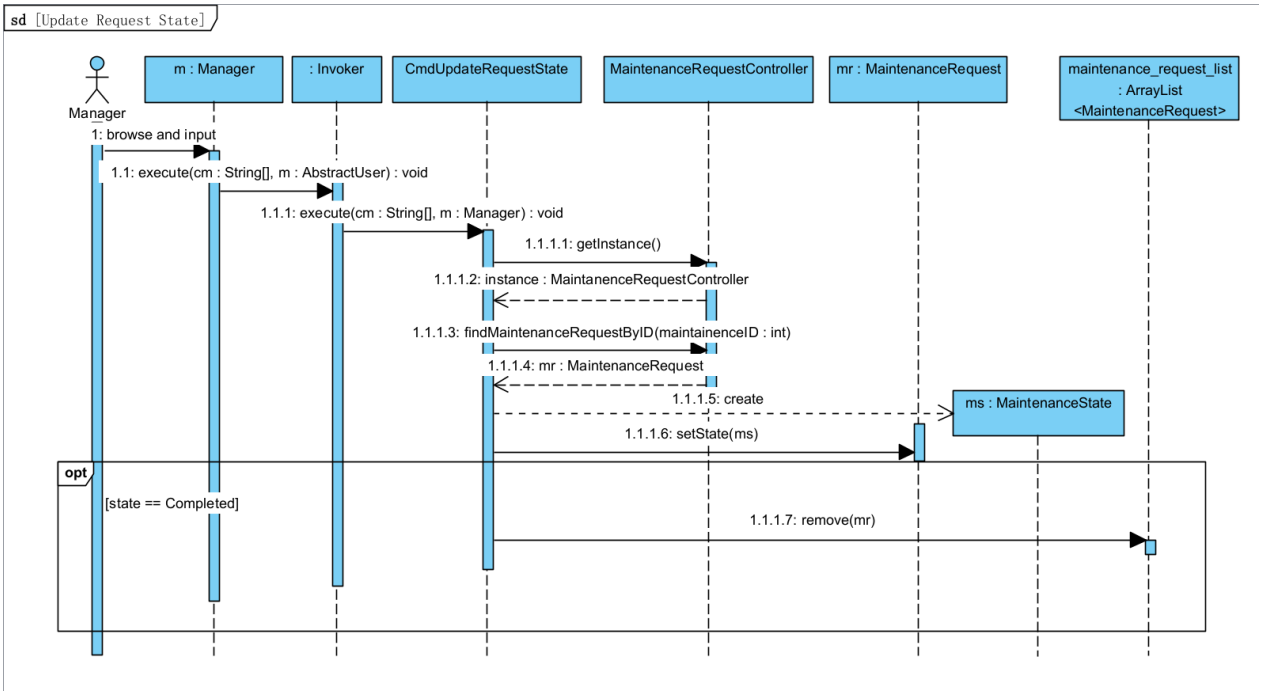

Figure 32: Update Request State

The **CmdUpdateRequestState** class is a **Manager** command. When a maintenance becomes **In Progress** or **Completed**, the manager can update the request state. Required information includes the maintenance ID and the new state.

CmdUpdateRequestState calls the **findMaintenanceRequestByID()** method in the **MaintenanceRequestController** class to find the request with the given maintenance ID. Then, it updates the request state to the new state. More specifically, a **MaintenanceState** object corresponding to the new state is created and assigned to the request.

In case the request is **Completed**, it will also be removed from the list.

5.6 Complaint Module

There is 1 **Resident** command and 1 **Manager** command in the Complaint module, which are **CmdSubmitComplaint** and **CmdHandleComplaint** respectively.

5.6.1 Submit Complaint

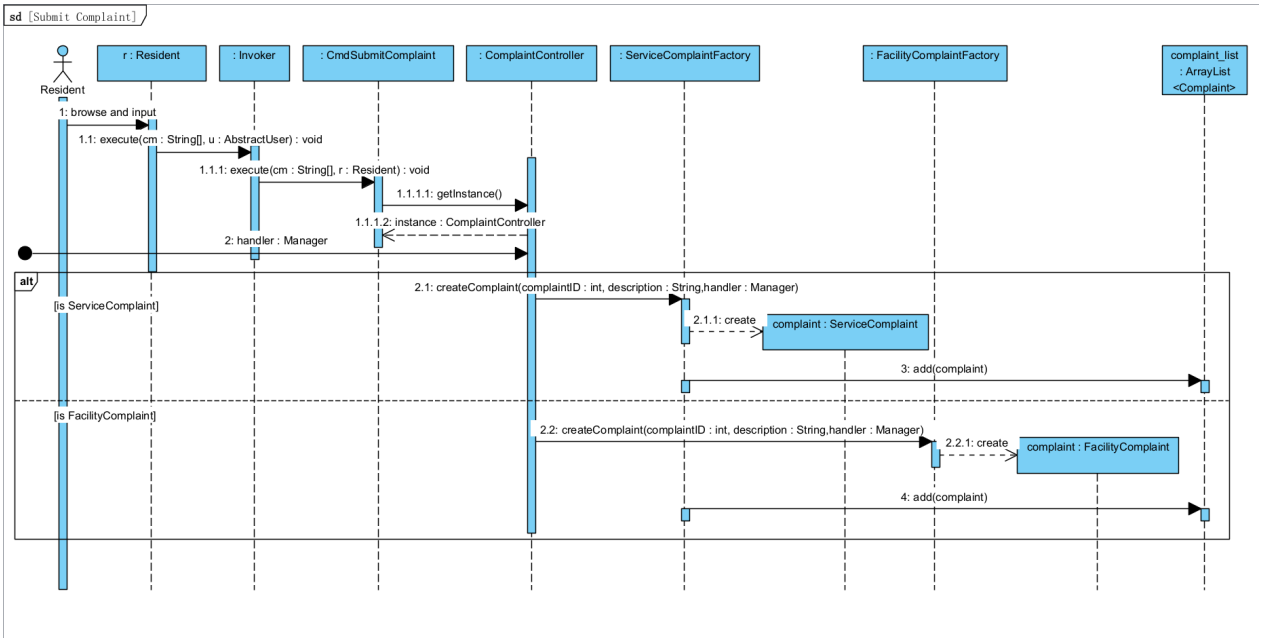

Figure 33: Submit Complaint

The **CmdSubmitComplaint** class is a **Resident** command. After inputing the complaint type and description, the request will be sent to **Invoker** and **CmdSubmitComplaint**.

Each complaint will be assigned a handler. Since there are two types of complaints, namely **ServiceComplaint** and **FacilityComplaint**, the **CmdSubmitComplaint** class delegate the object creation to the corresponding factory class, **ServiceComplaintFactory** and **FacilityComplaintFactory**.

The factory classes create a **Complaint** object with the provided information and append it to the **complaint_list**.

5.6.2 Handle Complaint

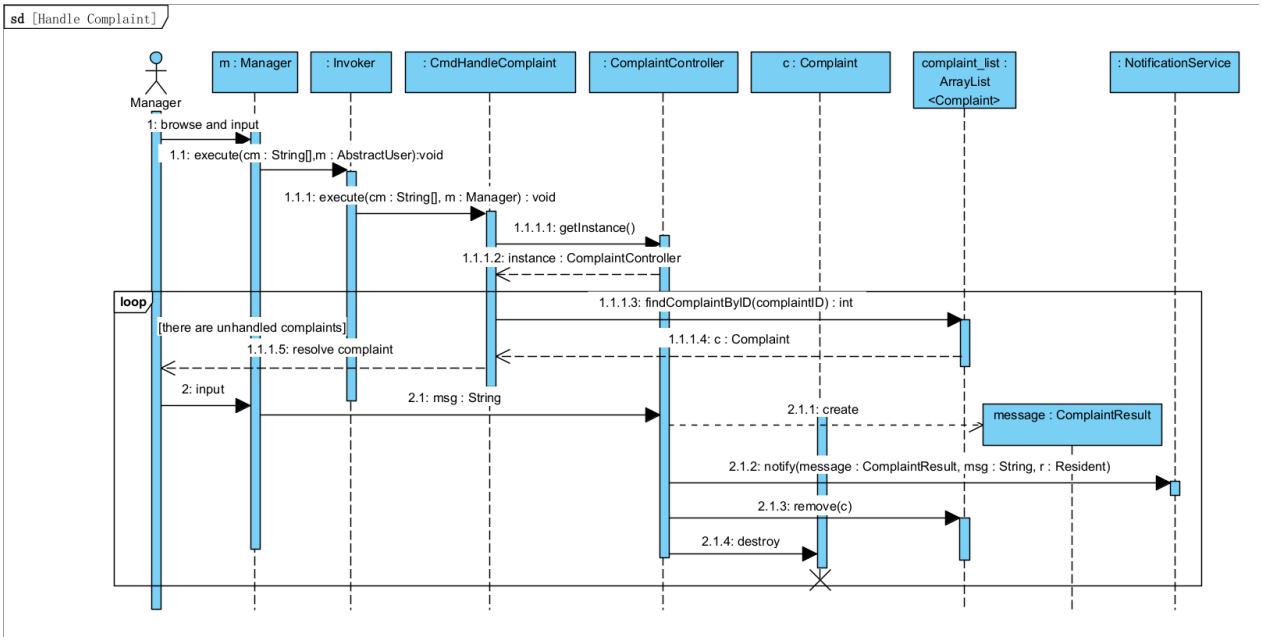

Figure 34: Handle Complaint

The **CmdHandleComplaint** class is a **Manager** command.

When the manager starts to handle a complaint, **CmdHandleComplaint** calls the **findComplaintByID()** method in the **ComplaintController** class to find the complaint with the given complaint ID. The system then prompts the manager to input the resolution.

After the manager inputs the resolution, the system will send a **ComplaintResult** notification to the resident who submitted the complaint. The complaint will be removed from the list and then discarded.

This process loops until there are no more complaints to handle.

5.7 Reservation Module

There are 2 **Resident** commands in the Reservation module, which are **CmdMakeReservation** and **CmdCancelReservation**.

Public places and services can be reserved on a first-come-first-served basis. Therefore, no verification is involved in the reservation process.

5.7.1 Make Reservation

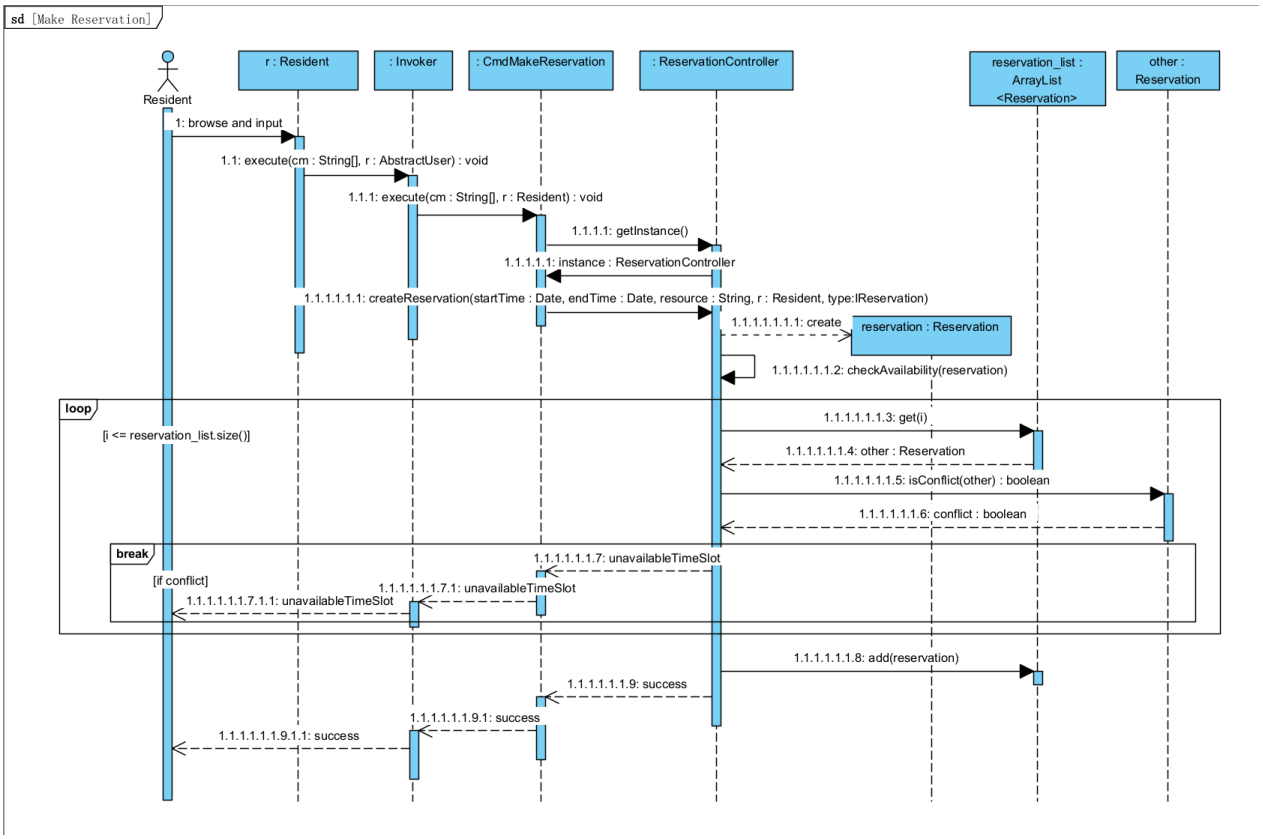

Figure 35: Make Reservation

The **CmdMakeReservation** class is a **Resident** command. The required information includes the start time, end time, description and the type of the reservation.

CmdMakeReservation calls the **createReservation()** method in the **ReservationController** class, which creates a **Reservation** object with the provided information.

It then calls **checkAvailability()** method to check if the reservation time slot is available. This method will iterate through all existing reservations to find if there is any overlap with

the new reservation. **Reservation** has a **isConflict()** method which returns **true** if two reservations share the same type and time slot.

There are two possible outcomes:

- If the time slot is available, the reservation will be appended to the **reservation_list** and a success message will be returned.
- If the time slot is not available, the method will return a **Unavailable time slot** error message.

5.7.2 Cancel Reservation

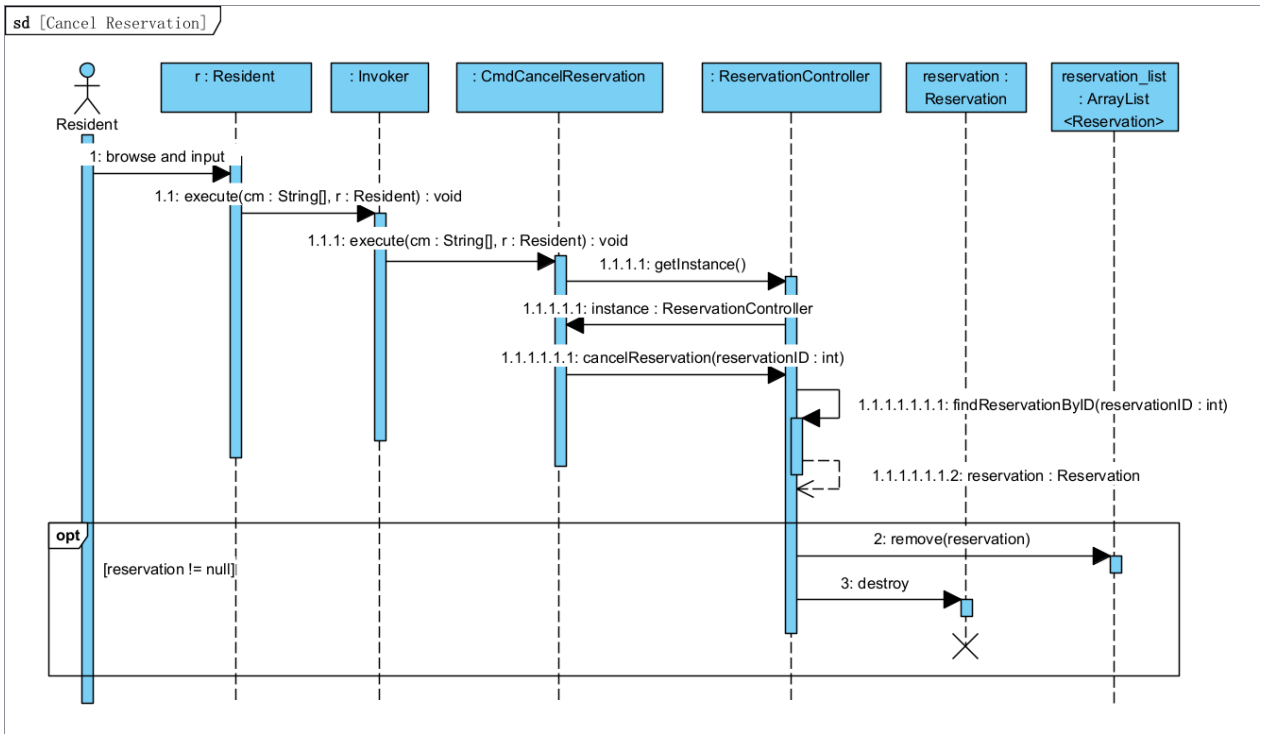

Figure 36: Cancel Reservation

The **CmdCancelReservation** class is a **Resident** command. The required information includes the reservation ID.

CmdCancelReservation calls the **cancelReservation()** method in the **ReservationController** class, which will find the reservation with the given reservation ID by **findReservationByID()** method.

Then, it removes the reservation from the list and discards it.

6 Prototype

6.1 Mobile App

6.1.1 Log-in

The Log-in interfaces for users are demonstrated in the following pictures. After entering the app, the user can choose different account type and log into different modes with their user id/name and password.

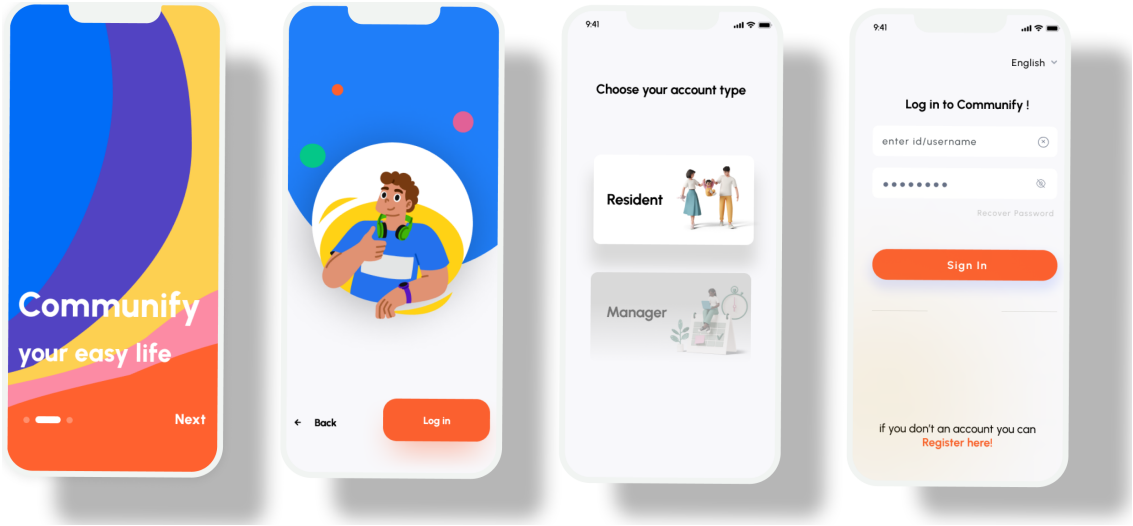

Figure 37: Log-in

6.1.2 Resident Mode

Home Page

The example image illustrates the basic layout of the home page. It can be seen that there are four integrated functions can be utilized by residents. After click each icon, it will link to the corresponding functional page. And the announcement is the part to release some information,notification,activities about community by managers, which help residents update the latest news.

Account Settings

Transitioning to the right screen, we see a profile-centric view that allows users to navigate through personal settings and features like Profile, Visitors, and more, with a prominent display of the user's profile picture and name at the top, reinforcing a personalized experience.

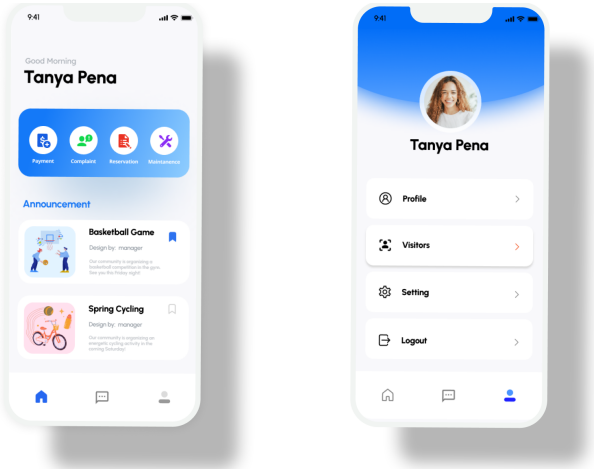

Figure 38: Home Page & Personal Account

Visitor Registration

In the "Account Settings" of the app, selecting the "Visitors" feature leads to a functional interface designed to streamline the visitor management process. The left screen illustrates a user-friendly form for the input of visitor details, including first and last names, contact information, and scheduled visit timing. Accompanied by a vibrant illustration, the form promotes an engaging user experience.

Upon submission, the right screen reveals a generated "Visitor Pass," which includes a QR code for easy verification, visitor ID, and the visit's date and time. The pass also specifies the time allotted for the visit, with a gentle reminder that the e-pass is valid for a limited duration.

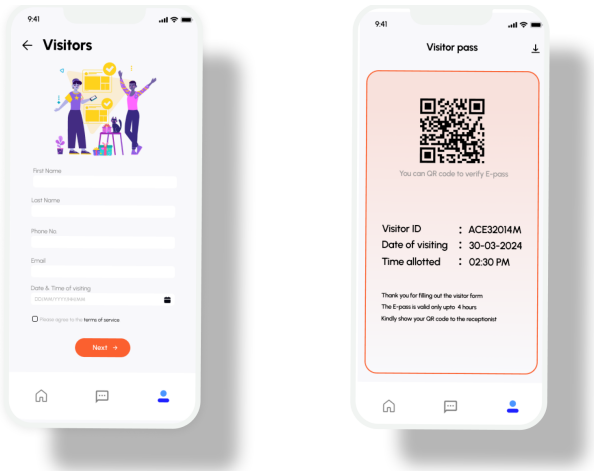

Figure 39: Visitor

Payment

Within the Payment section, users are presented with a clear breakdown of their monthly bills, categorized by services like property management, electric vehicle charging, and utilities. Each category includes a prompt to 'Pay Bill,' facilitating immediate transactions. Moving to the payment method selection, the app offers multiple modern options, including credit/debit cards and digital wallets. After payment, a success screen with payment details affirms the completion of the transaction, coupled with an option to view the receipt, demonstrating the system's emphasis on a secure and confirmatory payment process.

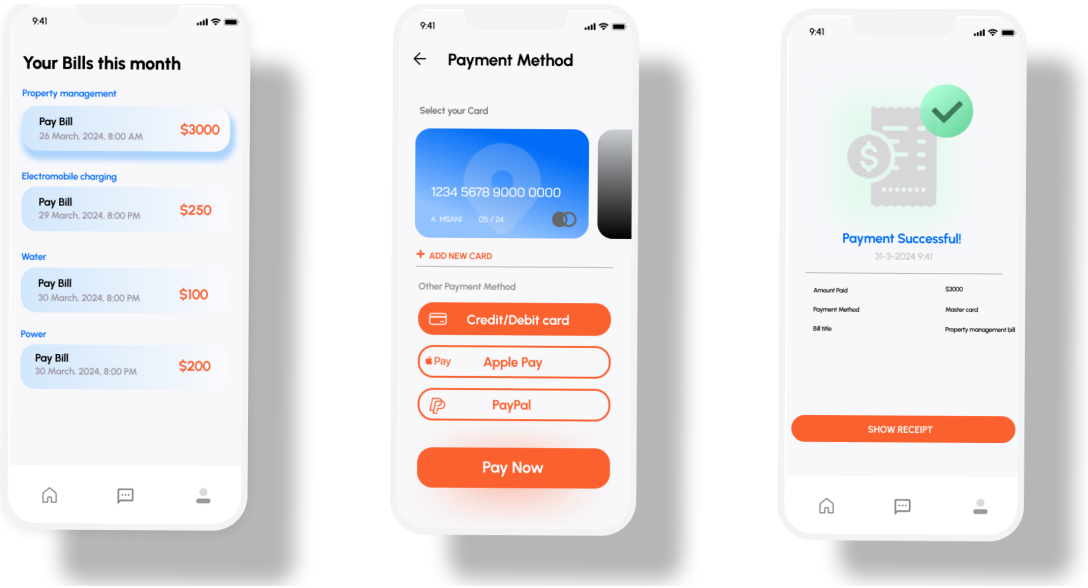

Figure 40: Payment

Complaint

The Complaint module presents a streamlined interface for users to report issues. Selectable complaint types, like 'facility' or 'service,' allow for categorical sorting. Users can enhance their reports with image uploads, providing visual context to the descriptions. An intuitive text box is available for elaborating on the problem, and voice recognition functionality suggests an advanced, accessible design. After submission, users receive a confirmation screen assuring them that the complaint is lodged and will be addressed, promoting a sense of user trust and process transparency.

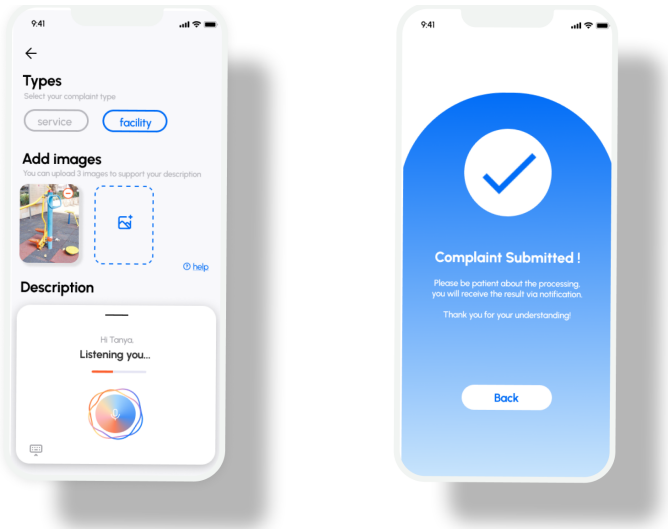

Figure 41: Complaint

Reservation

The Reservation feature facilitates booking of community amenities. Users can select a resource, such as a gym or restaurant, specify the number of people, and choose the date and time from a sleek calendar and clock interface. Upon completing the reservation, users are directed to a confirmation page detailing the booking, reinforcing the app's commitment to providing a hassle-free reservation experience. The use of vibrant icons and straightforward selectors underlines the app's focus on an efficient and pleasant user interface.

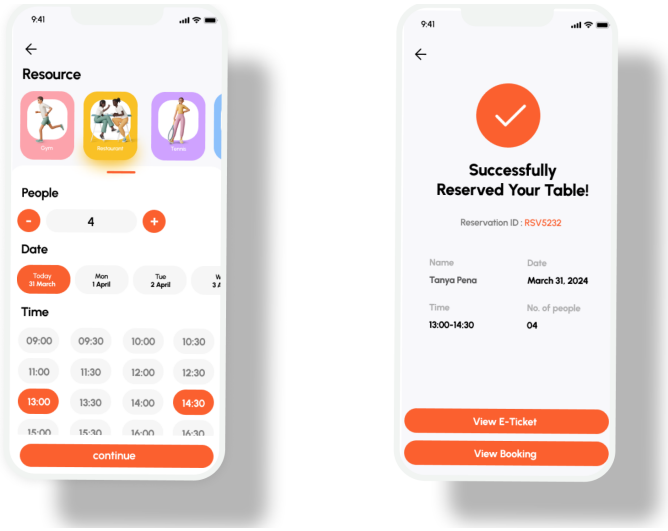

Figure 42: Reservation

Maintenance

This prototype introduces a maintenance overview dashboard where users can track the status of their various maintenance requests. It displays a count of total, pending, ongoing, and completed maintenance tasks, along with a severity scale for quick prioritization. The interface provides a succinct summary of each issue, categorized by type, with options to view details, track progress, or cancel requests. The visual layout is designed for ease of use, with clear indicators that guide users through their maintenance management tasks.

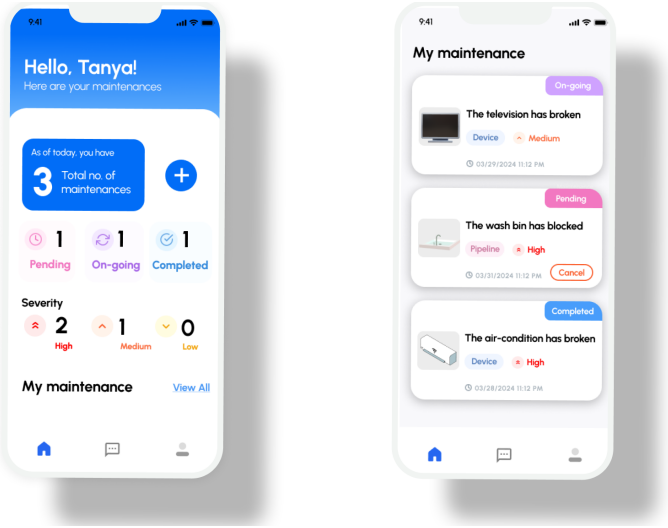

Figure 43: Maintenance

Notification

The Notifications feature of the app aggregates all user alerts in one place, with a clean and straightforward interface. Users can easily track updates on bill payments, complaint resolutions, maintenance tasks, reservations, and visitor arrivals, each distinguished by colorful icons. This design supports quick scanning and prioritization, keeping users informed and engaged with timely and organized information.

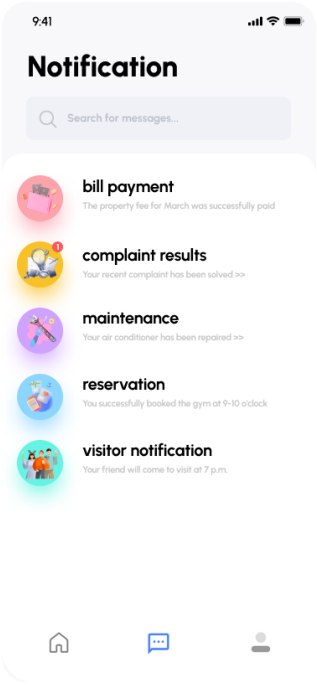

Figure 44: Notification

6.1.3 Manager Mode

Home Page

Upon successful authentication, the user is presented with a "Manager Mode" interface that is designed to facilitate seamless task management and oversight. The main dashboard prominently displays a task completion gauge, offering at-a-glance insight into progress with a concise "50% Completed" indicator. Below, key managerial functions—such as Complaints, Maintenance, and Visitor management—are accessible via intuitive icons, streamlining navigation. Adjacent to these, a "Work Statistics" chart provides a visual breakdown of weekly activity, enabling managers to monitor operational tempo and adjust resources accordingly. This prototype encapsulates a user-centric design philosophy, ensuring that essential management tools are within easy reach for efficient day-to-day operations.

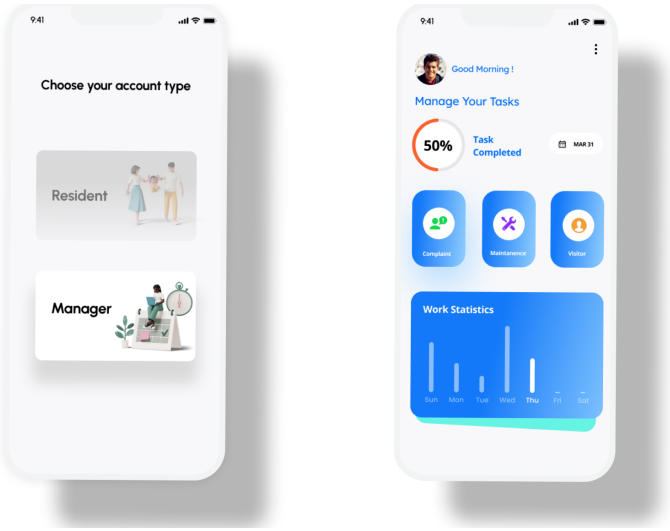

Figure 45: Manager mode

Complaint Management

As a manager taps on the "Complaint" option within the app, they are navigated to a comprehensive listing of current complaints, such as the example highlighted for "Regarding the broken arm of facility." This screen showcases a well-organized layout that simplifies the monitoring process, with options to filter complaints by categories like 'Facility' or 'Service'. Selecting a particular complaint unfolds detailed information, ensuring the manager has full context and can promptly respond. The details include the complaint's description, submitter's contact, and attached images for better assessment. Additionally, resolution options are provided to either address the complaint or mark it as resolved, with predefined reasons for complaint rejection to streamline the workflow and enhance decision-making efficiency.

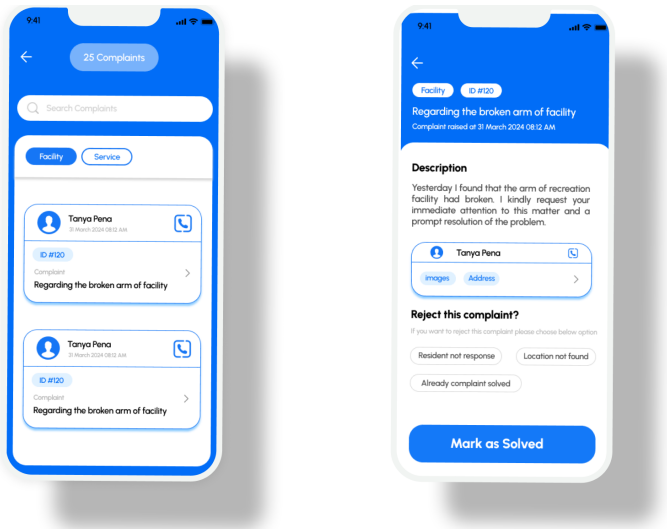

Figure 46: Complaint Management

6.2 Web Dashboard

6.2.1 Main Functions

In addition to the mobile app for residents, our app also has the web dashboard version for property managers (Figure 47). There are five main functional sections included in the web dashboard page, which are the **navigation bar, daily overview, comments section, repair request, and visitor profile**. In the sample figure, we show the overview page of the web dashboard. It shows the daily overview as well as the overview of reviews, visitors, and maintenance requests, where the reviews section contains the complaint information. The web dashboard allows the manager to have a clear picture of the community and to interact with the residents in every aspect through the **Notification** function.

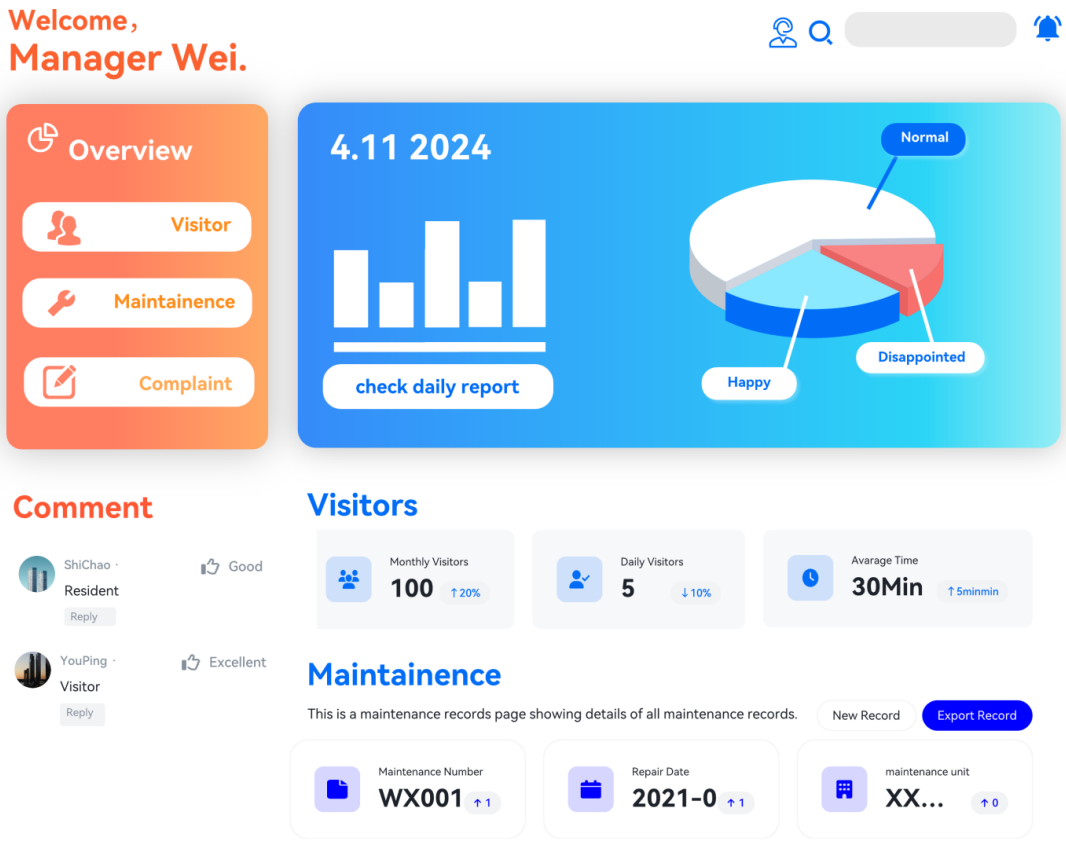

Figure 47: Web Dashboard

7 Conclusion and Reflection

7.1 Conclusion of the Community & Management App

The motivation behind our **one-stop Community Service and Management App** stems from the desire to create a connected, efficient, and transparent community. By providing convenient access to various services, simplifying management processes, enhancing community engagement, and ensuring transparency, we aim to empower residents and community managers alike. Our app serves as a catalyst for positive change, fostering a vibrant and harmonious community where residents can thrive and actively contribute to the betterment of their living environment.

7.2 Weekly Activity Log

This is our weekly activity log for reference.

Week	Weekly activity log	Completed By
Week 1	Recruit Project Memebers to Form a Team.	All
Week 2	Select the topic.	All
Week 3	Confirm the desirable core functions.	All
Week 4	Confirm other functions to enlarge the function scale (add increments).	All
Week 5	Use case diagram + discussion.	Wan Zimeng, Wang Ying + All
Week 6	Use case description.	Wang Ying, Wan Zimeng
Week 7	Class diagram design + discussion.	LIU Hengche,Chen Yihuan + All
Week 8	Sequence diagram design + discussion	Wang Fan, Gao Nanjie + All
Week 9	Midterm preparation.	/
Week 10	User Interface making (mobile+dashboard).	Chen Yihuan, Gao Nanjie
Week 11	Writing Report.	All
Week 12	PowerPoint making.	All
Week 13	Formatting, preparing for the presentation, finishing the report.	All

Our workload distribution is illustrated as follows. Each of our group mates is responsible

for the report and the presentation PowerPoint making.

SID	Name	Position	Work allocation	Contribution
57854946	GAO Nanjie	Project Manager	Requirement Analysis Sequence Diagram Background	100%
57854262	WANG Ying	Assistant Project Manager	Requirement Assembly Use Case Diagram Motivation & Process Model	100%
57222904	WAN Zimeng	Designer	Background Research Use Case Diagram Abstraction	100%
57854004	WANG Fan	Designer	Requirement Analysis Sequence Diagram Alternatives	100%
57853935	CHEN Yihuan	Designer	Prototype Visualization Class Diagram Risk and Constraints	100%
57854329	LIU Hengche	Designer	Requirement Analysis Class Diagram Scope	100%

7.3 Achievements

Our team followed the Agile methodology to develop the app, which allowed us to adapt to changing requirements and deliver a minimum viable product (MVP) as quickly as possible. After initial research and planning, we divided our system into four main components: **User Management**, **Core Service**, **Interfacing with External Services**, and **UI/UX Design**. Based on the importance of each component, we decided to focus on these areas one at a time, in that order.

User Management was the first component we developed, as it built the foundation for the rest of the app. Core Service was the next component we focused on, as it involved everyday tasks and services, including maintenance requests, visitor management, and complaint resolution. We then worked on Interfacing with External Services to integrate payment gateways and notification services. Finally, we focused on UI/UX Design to ensure a seamless and intuitive

user experience.

Throughout the development process, we conducted regular meetings, code reviews, and testing to ensure the quality and functionality of our app. We also incorporated feedback from stakeholders to improve the app's usability and features. By following this iterative approach, we were able to deliver a functional and user-friendly app that fulfilled the requirements allocated from our client.

7.4 Further Development

Despite the progress we have made, there are still areas for further development and improvement. Some of the key areas we identified include:

- **Enhanced Security Features:** Implementing additional security measures, such as two-factor authentication and encryption, to protect user data and privacy.
- **Data Analytics and Reporting:** Implementing data analytics and reporting features to provide insights into community trends, service usage, and resident feedback.
- **Community Forums and Social Features:** Adding community forums and social features to facilitate communication and collaboration among residents.
- **Localization and Multi-language Support:** Providing multi-language support and localization features to cater to residents from diverse cultural backgrounds.
- **Accessibility Features:** Implementing accessibility features, such as screen readers and voice commands, to make the app more inclusive and user-friendly.

We firmly believe that by addressing these areas and continuously improving our system, we can create a more robust and comprehensive Community Service and Management App that benefits residents and community managers alike.

References

[1] H. Mu and S. Jiang, "Design patterns in software development," 2011 IEEE 2nd International Conference on Software Engineering and Service Science, Beijing, China, 2011, pp. 322-325, doi: 10.1109/ICSESS.2011.5982228.

[2] Richard N. Taylor, “Software Architecture and Design,” in Handbook of Software Engineering, Springer Nature Switzerland AG 2019, doi: 10.1007/978-3-030-00262-6.

[3] “Basic Design Principles in Software Engineering,” 2012 Fourth International Conference on Computational and Information Sciences, August 17-19, 2012, doi: 10.1109/ICCIS.2012.91.